aardvark

A Mantine-powered static site generator. Author in Markdown, build to static HTML with interactive React islands.

https://aardvarkdocs.com

Generated 2026-08-19 16:11 UTC

Contents

Why Aardvark? 12

Introduction 16

Installation 19

Project structure 21

Templating & data 22

Markdown reference 25

Twoslash 43

Build-time Python 45

Components & snippets 52

Custom components 56

Custom snippets 60

Block components 62

Custom CSS & JS 64

Navigation menus 66

Tables 70

Overview 72

Wide mode 74

Full mode 75

TOC-only mode 76

Uncapped mode 78

Theme & customization 81

Light & dark mode 89

Landing page template 91

Search 92

Content Reach 101

Analytics & ratings 106

Reader surveys 111

Definitions & glossary 117

Accessibility 122

Versioning 127

Password protection 129
Deploy Aardvark 133
Deployment 134
CLI reference 137
Generated files 148
Deep nesting 155
Level 2 156
Level 3 157
Level 4 158
Level 5 159
AI Features 160
AI assistant & analytics 162
Aardvark cloud gateway 169
Build-time AI 173
Docs Quality Checks 174
Self-hosting & MCP 180
Agent readiness 184
Agent discovery 191
Web Bot Auth 196
Plans & pricing 198
AI model rates 204
Aardvark Extras 217
API Fields 218
Banner 224
Changelog 226
CodeGroup 232
Component libraries 237
Examples 243
Gitfolder 247
Include 253
Map 255

OpenAPI 259
Panel 271
Prompt 275
Target 279
Taxonomy 281
Update 291
Visibility 296
Layout 300
AppShell 302
AspectRatio 307
Box 311
Center 315
Collapse 319
Container 323
Flex 327
Grid 331
Group 335
Marquee 341
Paper 345
Portal 349
Scroll Buttons 353
ScrollArea 357
Scroller 361
SimpleGrid 365
Space 371
Splitter 374
Stack 378
Inputs 383
AlphaSlider 385
AngleSlider 388
Checkbox 392

Chip ... 397
ColorInput ... 402
ColorPicker ... 407
DateInput ... 411
DatePickerInput ... 414
DateTimePicker ... 417
Dropzone ... 420
Fieldset ... 423
FileInput ... 427
FocusTrap ... 432
Form ... 436
HueSlider ... 439
Input ... 442
JsonInput ... 446
MaskInput ... 451
NativeSelect ... 457
NumberInput ... 462
PasswordInput ... 469
PinInput ... 474
Radio ... 478
RangeSlider ... 483
Rating ... 488
RichTextEditor ... 492
SegmentedControl ... 494
Slider ... 499
Switch ... 503
Textarea ... 508
TextInput ... 514
TimeInput ... 520
Combobox ... 523
Autocomplete ... 524

Combobox 529
MultiSelect 533
Pill 539
PillsInput 543
Select 548
TagsInput 554
TreeSelect 559
Buttons 564
ActionIcon 565
Button 572
CloseButton 584
CopyButton 588
FileButton 592
UnstyledButton 596
Navigation 601
Anchor 602
Breadcrumbs 607
Burger 610
NavLink 614
Pagination 619
Steps 623
TableOfContents 627
Tabs 631
Tree 638
Feedback 645
Callout 646
Loader 650
Navigation progress 653
Notification 655
Notifications 660
Progress 662

Ring progress ... 666
Semi-circle progress ... 670
Skeleton ... 674
Overlays ... 678
Affix ... 679
Dialog ... 683
Drawer ... 687
FloatingIndicator ... 693
FloatingWindow ... 696
HoverCard ... 700
LoadingOverlay ... 705
Menu ... 709
Modal ... 714
Modals manager ... 720
Overlay ... 723
Popover ... 727
Spotlight ... 732
Tooltip ... 735
Transition ... 740
Data Display ... 744
Accordion ... 745
AreaChart ... 751
ArticleCard ... 754
Avatar ... 760
BackgroundImage ... 765
Badge ... 768
BarChart ... 773
Card ... 775
Carousel ... 787
ColorSwatch ... 791
DonutChart ... 795

Icon ... 798
Image ... 805
Indicator ... 811
Kbd ... 816
LineChart ... 820
NumberFormatter ... 823
OverflowList ... 827
RollingNumber ... 831
Spoiler ... 836
Timeline ... 840
Typography ... 845
Blockquote ... 846
Code ... 850
CodeBlock ... 854
Divider ... 857
Highlight ... 862
List ... 866
Mark ... 870
Table ... 874
Text ... 879
Title ... 886
Typography ... 891
VisuallyHidden ... 895
Community Components ... 898
Clock ... 901
ContextMenu ... 905
Picker ... 909
Rings progress ... 912
Led ... 915
Lightbox ... 919
Select stepper ... 922

Window 925
QR code 929
Spinner 932
SplitPane 936
Compare 939
Mask 943
Book 947
DepthSelect 950
Scene 954
Flip 958
Parallax 962
Reflection 967
JsonTree 971
LensSelect 976
ListViewTable 980
Audio 985
Video 988
Onboarding 992
DataTable 996
BlockNote 1000
Community Marquee 1002
Text Animate 1006
Border Animate 1010
Aardvark Gateway API 1014
Support 1089
Anatomy of a Mantine island 1093
Designing the honest AI meter 1094
Introducing Ask AI, the reader assistant 1095
Multi-language docs in practice 1096
vark 0.9 release roundup 1097
Why Aardvark rebuilds every page (for now) 1098

Changelog ... 1099
Blog ... 1101
Build-time accessibility contrast audit ... 1102
Ask-AI reader assistant ... 1103
Generate the CLI reference from `vark --help` ... 1104
Standardized code-block Copy & Download buttons ... 1105
Faster dev-loop rebuilds ... 1106
Interactive Mantine islands ... 1107
Multi-language sites ... 1108
Index OpenAPI descriptions for search ... 1109
Social unfurl cards (Open Graph) ... 1110
Built-in Components ... 1111
How do I add or remove seats? ... 1113
What happens to my add-on seats when I change plans? ... 1114
What gets used first — my allowance or my prepaid balance? ... 1115
How does annual billing save money? ... 1116
What is auto top-up? ... 1117
What if I fire a lot of AI requests at the same time? ... 1118
How does Enterprise billing work? ... 1119
What if a subscription payment fails? ... 1120
How do seats work? ... 1121
What if I don't use all of it — does it roll over? ... 1122
What's the most I could be billed in a month? ... 1123
Why do I have to confirm a checkbox to raise my overflow cap past 2× my included 1124
What are my options when the included AI runs out? ... 1125
Do I need my own model API keys? ... 1126
What happens to my prepaid balance if I cancel, downgrade, or a payment fails? ... 1127
What if our published prices change after I subscribe — does my bill go up? ... 1128
What if I want to raise the cap again later — do I re-confirm every time? ... 1129
Can I remove my saved card while I'm on a subscription? ... 1130
What happens when I run out of seats? ... 1131

Can I still self-host on Pro or Business? 1132
Is the cutoff an exact-to-the-penny $0 guarantee? 1133
What if I schedule a downgrade and then change my mind? 1134
Why did my AI pause with an “unusual usage” message? 1135
What happens when I use up my included AI for the month? 1136
What exactly counts against the included AI? 1137
What happens when I cancel? 1138
When do plan changes take effect? 1139

Markdown in · agent-ready out

The first word in documentation

Aardvark is a Mantine-powered static site generator. Author in Markdown, build to fast static HTML with interactive React islands.

Get started

```
zsh — my-docs
$ vark new my-docs
  scaffolded project
$ vark dev --port 8000
  built 24 pages · serving http://127.0.0.1:8000
  live-reload ready
```

⌘K search OpenAPI sitemap.xml llms.txt OG cards i18n dark mode

45+

built-in components

16

analytics integrations

300+

AI models, every major lab

1

binary to deploy

Batteries included

Everything a docs site needs

From templating to search to deploy — the parts you'd otherwise wire together yourself, built in.

Markdown in, HTML out

Author in Markdown, build to fast static HTML — pretty URLs, a TOC, and a clean default theme.

Real Python templating

Logic lives in template tags and is actual Python — pull from JSON, YAML, and CSV data files.

Generate any file

Build-time Python emits whole page sets — or any downloadable file: CSV, JSON, YAML, even binaries.

Built-in ⌘K search

A full-text search index and a fast command-palette modal ship with every build.

OpenAPI try-it

Drop in a spec and get interactive "try it now" reference pages.

Ships as one binary

Compile the whole tool to a single executable with Nuitka. Node only at build time.

Plug into your stack

Works with the tools you already use

Drop-in analytics and integrations — 16 platforms, each switched on with one config block. No script tags to hand-roll, no consent plumbing to wire.

Google Analytics Amplitude PostHog Segment Mixpanel Microsoft Clarity Plausible Heap Hotjar LogRocket Google Tag Manager Clearbit Fathom Hightouch Adobe Analytics Pirsch

Author once

Markdown that does more

Write content the way you always have — then reach for real Python: inline in a page, or in a generator script that emits whole page sets and downloadable files.

Loop and template with {% %} Python tags Pull from JSON, YAML, and CSV data files Emit CSV, JSON, YAML — or any file — from a generator

Read the build-time Python guide

page.md

```
---
title: Pricing
---
# Pricing
There are {% data.products.count %} plans.
{% component('Button', children='Start free') %}
```

AI, built in

An assistant your readers can install

Ship a docs assistant that adds to the home screen, answers from your own content, and reads what readers hand it — backed by hundreds of models and billed only when it runs.

Installable assistant

A progressive web app — readers add it to a phone or desktop and launch straight into full-screen chat. No app store.

Multimodal questions

Attach images, PDFs, and text or code — by paperclip, drag-and-drop, or paste — and ask about them in context.

Instant, no-backend search

⌘K scores a prebuilt index right in the browser — no Algolia, no crawler, no server to keep running.

Free to use. Aardvark builds a plain static site you can host anywhere — Netlify, Vercel, S3, or your own box. You only pay for AI, and only when it runs: **pay-as-you-go**, metered per token through the cloud gateway.

300+ models, from every major lab

OpenAI Anthropic Google xAI Meta Mistral AI DeepSeek Cohere Qwen Perplexity Microsoft Amazon

Aardvark

Aardvark is a documentation-focused static site generator. You author in Markdown and build to static HTML. Interactive UI is delivered as **islands**: Aardvark emits the HTML, and Mantine + your own React components render in the browser from a single bundled runtime.

This very site is built with Aardvark.

> 🚀 **Build your first site in a minute**
>
> Markdown in, fast static HTML out — with interactive Mantine islands. No build config to write.

Why Aardvark

- **Markdown in, HTML out** — pretty URLs, a TOC, and a clean default theme.
- **Real Python templating** — logic lives in `{% %}` tags and is actual Python, with your structured data exposed as `data.<file>.<prop>`.
- **Every Mantine component** — embed any of the Mantine UI components (and your own React snippets) directly from Markdown; they render as islands.
- **Batteries included** — built-in ⌘K search, a built-in "Ask AI" assistant, Google Analytics with page-rating events, OpenAPI "try it now" reference pages, and auto-generated `sitemap.xml` + `llms.txt`.
- **Optional build-time AI** — when enabled, generate frontmatter, example API responses, and a skill-creation plan from your docs.
- **Ships as one binary** — compile the tool to a single executable with Nuitka. Node is only needed at build time to bundle the islands.

How it fits together

1. You write Markdown in `content/`, structured data in `data/`, and optionally custom React components in `snippets/`.
2. `vark build` runs the `{% %}` Python templating, renders Markdown to HTML, collects every `{% component(...) %}` call into island mount points, and bundles the referenced Mantine/snippet components with esbuild.
3. The output in `build/` is plain static HTML + one JS/CSS bundle, ready to host anywhere.

Explore the docs

Installation

Install the CLI and you're ready to build.

Quickstart

Scaffold and build your first site in a minute.

Component gallery

Live examples of every Mantine component, callable straight from Markdown.

Quickstart

Once you've installed Aardvark, you can scaffold, preview, and build your first site in a few commands.

1. Scaffold a site

```
vark new my-docs
cd my-docs
```

This creates a project with a starter page, a data file, the editable default theme, an example snippet, and a `package.json` for the islands.

2. Install the islands dependencies

```
npm install
```

This installs React, Mantine, and esbuild, which Aardvark uses to bundle the components you embed.

3. Develop with live reload

```
vark dev --port 8000
```

`vark dev` builds the site, serves it on `http://127.0.0.1:8000`, opens that URL in your browser, watches your source files, and reloads the browser on every change. Pass `--no-open` if you'd rather open the tab yourself.

4. Build for production

```
vark build      # outputs static HTML to ./build
```

Create your first page

An Aardvark page is Markdown with optional YAML frontmatter. Logic is real Python inside `{% %}` tags:

```
---
title: Hello
---

# Hello

This site has {% len(components) %} Mantine components available.

{% component('Button', children='Click me', color='blue') %}
```

That's it. Continue with Project structure or jump to Templating & data.

Installation

Prerequisites

Tool	Version	Why
Node.js + npm	Node ≥ 20.19	Bundles the Mantine/React islands at build time

The `vark` binary is self-contained — it ships its own Python runtime, so there's nothing else to install. Node is only needed to build the islands JS. You can skip it with `vark build --no-bundle` (components then render as inert placeholders).

Install with Homebrew (macOS)

The quickest way on a Mac — installs a self-contained binary, no Python required:

```
brew tap aardvarkdocs/tap
brew install aardvark
vark --version
```

The Homebrew package is named `aardvark` (after the project), but it installs the CLI as **`vark`** — shorter to type. (`aardvark` is also installed as an alias, so either name works.) The one-time `brew tap` registers the formula repository; after that `brew install aardvark` / `brew upgrade aardvark` work like any Homebrew package. If you'd rather not tap, the single-command equivalent is `brew install aardvarkdocs/tap/aardvark`.

Upgrade later with `brew update && brew upgrade aardvark`. Interactive Mantine islands still need Node at build time (`brew install node`); without it, build with `vark build --no-bundle`.

Windows

Download the prebuilt Windows binary from the latest release:

1. Grab `aardvark-<version>-windows-x86_64.zip`.
2. Unzip it and move `vark.exe` somewhere on your `PATH` (e.g. a folder you've added under **Settings → System → About → Advanced system settings → Environment Variables**).
3. Open a new terminal and run `vark --version`.

The binary is **unsigned**, so the first launch may show a SmartScreen warning ("Windows protected your PC") — click **More info → Run anyway**. As on every platform, Node.js is only needed at site-build time for interactive islands; `vark build --no-bundle` works without it.

Linux

Download the matching tarball from the same release page (`aardvark-<version>-linux-x86_64.tar.gz` or `…-linux-aarch64.tar.gz`):

```
curl -fsSL
https://github.com/aardvarkdocs/homebrew-tap/releases/download/v<version>/aardvark-<version>
-linux-x86_64.tar.gz \
  | tar xz && sudo mv vark /usr/local/bin/
vark --version
```

The prebuilt binary is **glibc-linked** (not musl/Alpine). Release tarballs target **glibc ≥ 2.39** (Ubuntu 24.04 class and newer distros). Node.js is only needed at site-build time for interactive islands.

The binary bundles the default theme and the islands runtime, so it works out of the box. Node is still required on the build machine to bundle interactive islands — see Deployment for details.

AI features

Aardvark's AI features are built into the binary — there's nothing extra to install. Turn them on in `aardvark.config.yaml` (the `ai:` block) and provide a key. See AI features for the overview, Build-time AI for the build-time enrichment options, and Cloud gateway for the hosted reader assistant.

Project structure

A scaffolded Aardvark project looks like this:

```
my-docs/
  aardvark.config.yaml      # site metadata, tabs, integrations, AI flags
  content/              # *.md pages  ->  pretty URLs
  data/                 # *.json *.yaml *.csv  ->  data.<file>.<prop>
  themes/vark/          # the docs theme (full, editable HTML source you own)
  templates/            # optional: override a single theme file (else omit)
  snippets/             # your custom *.jsx / *.tsx React components
  openapi/              # OpenAPI specs (optional)
  *.css *.js            # project-root assets -> copied, fingerprinted, linked
  static/  public/      # public assets -> copied and fingerprinted (optional)
  package.json          # islands deps (react, @mantine/*, esbuild)
  build/                # generated output
```

What each directory does

- **`content/`** — One `.md` file per page. `content/index.md` → `/`, `content/guide/intro.md` → `/guide/intro/`. See Templating & data.
- **`data/`** — JSON, YAML, and CSV files. Each file becomes `data.<filename>` in your pages; a CSV becomes a list of row objects.
- **`themes/vark/`** — Your editable copy of the active theme. `default.html` is the page layout, rendered by the same `{% %}` engine; a project-local `themes/<name>/` wins over the bundled theme. Add more layouts (e.g. `landing.html`) and select them per page with `pagetype:`. See Theme & customization.
- **`templates/`** — Optional. Drop one file here to override just that file of the active theme (a lightweight alternative to forking the whole `themes/vark/`).
- **`snippets/`** — Your own React components, usable from Markdown by filename. See Components & snippets.
- **`openapi/`** — OpenAPI specs you render inline with the `{% openapi %}` directive — a whole spec, or a single-endpoint slice — on an ordinary Markdown page. See OpenAPI.
- **Root `*.css` / `*.js`** and **`static/` / `public/`** — copied into the build with per-build filenames such as `/img/logo-<sha>.svg`; keep authoring links as `/img/logo.svg`. See Custom CSS & JS.

Output

`vark build` writes everything to `build/`: one HTML file per page (as `<url>/index.html`), the bundled islands JS/CSS under `_aardvark/`, your fingerprinted static assets, plus generated files (`sitemap.xml`, `llms.txt`, `llms-full.txt`).

Templating & data

Logic in Aardvark pages is **real Python**, written inside `{% %}` tags.

The two kinds of block

A block that is a **single expression** is evaluated and its result is printed:

```
Today's answer is {% 6 * 7 %}.
```

A block that is one or more **statements** runs via `exec`; it writes to the page with `page.print(...)` — the write-mirror of `page.get()`:

```
{%
for fruit in ["apples", "pears", "plums"]:
    page.print("- ", fruit, "\n")
%}
```

All blocks on a page share one namespace, so a variable set early is available later.

Data files

Drop `.json`, `.yaml`, or `.csv` files in `data/`. Each becomes `data.<stem>`:

- A JSON/YAML object is reachable as `data.file.property`.
- A CSV becomes a **list of row objects** keyed by the header row.

This site ships `data/products.yaml`. Here it is, live:

> There are **3** products. The first is **Sticker Pack** at $8.

A loop over the same data:

- Sticker Pack ($8)
- Enamel Mug ($18)
- Zip Hoodie ($55)

The Markdown for that loop was:

```
{%
for p in data.products.items:
    page.print(f'- {p.name} (${p.price})\n')
%}
```

What's in scope

Name	What it is
`data`	Your `data/` files (`data.file.prop`)
`site`	`site:` block from `aardvark.config.yaml`
`config`	The full configuration object
`page`	This page's front matter — `page.get(key, default)` reads it; `page.print(*strings)` writes to the page
`component(name, **props)`	Embed a React island. `component('library', 'Name', …)` reaches a theme component library; `component('aardvark', 'tag', …)` reaches a built-in by its tag name (e.g. `component('aardvark', 'card', …)`) so you can build one in a `for` loop. See Components.
`snippet(name, **props)`	Alias of `component` for your `snippets/`
`asset(path)`	Return the build's fingerprinted URL for a static asset. Use it when constructing URLs dynamically; literal asset URLs are rewritten automatically.
`components`	Sorted list of every registered component name
`print(*args)`	Lower-level page output (always available; the only form inside custom-component bodies, where `page` isn't in scope)

Showing literal syntax

To display `{% %}` without running it, wrap it:

```
{% raw %}
this {% will_not_run() %} is shown verbatim
{% endraw %}
```

Headings and anchor links

Every heading (levels 1–4) gets a stable `id` and a permalink. Hover a heading and a link icon fades in to its right — click it to jump to that section and put the anchor in your address bar to share. The `id` defaults to a GitHub-style slug of the heading text (`## Heading anchors` → `#headings-and-anchor-links`), and repeated headings are de-duplicated (`#setup`, `#setup-1`, …).

To pin a short, stable anchor yourself, add `{#custom-id}` at the end of the heading line:

```
## Configuration options {#config}
```

links as `#config` (the `{#…}` marker is removed from the rendered heading). Custom ids keep working even if you later reword the heading, so existing links don't break.

Page layout modes

A page's `mode` front matter controls its layout — toggling the left nav, the right-hand TOC, and the content width:

```
---
title: Release dashboard
mode: wide
---
```

The options are `wide`, `full`, `toc-only`, and `uncapped` (omit `mode`, or use `default`, for the standard nav-plus-TOC layout). See **Layout modes** for the full table and a live demo of each.

Markdown reference

Aardvark renders Markdown with markdown-it, following the CommonMark spec with **tables**, **strikethrough**, **footnotes**, **smart typography**, and **automatic links** turned on. This page shows the source for every element next to its live result — copy any block straight into your own pages.

Headings

Six levels, from `#` to `######`:

```
# Heading level 1
## Heading level 2
### Heading level 3
#### Heading level 4
##### Heading level 5
###### Heading level 6
```

renders, live:

Heading level 1

Heading level 2

Heading level 3

Heading level 4

Heading level 5

Heading level 6

Levels 1–4 get a stable `id`, a permalink that fades in when you hover, and an entry in the **On this page** table of contents; levels 5 and 6 render but stay out of the TOC. (Because the six lines above are real headings, you will see levels 1–4 appear in this page's TOC.) Pin a custom id by ending a heading line with `{#my-id}` — see Templating & data for more.

Paragraphs and line breaks

Separate paragraphs with a blank line. A single newline just soft-wraps. For a hard line break, end a line with two trailing spaces or a backslash `\`:

```
First paragraph.

Second paragraph,\
forced onto a new line within the same paragraph.
```

renders, live:

First paragraph.

Second paragraph,
forced onto a new line within the same paragraph.

Text formatting

```
**Bold** and __bold__, *italic* and _italic_, ***bold italic***,
~~strikethrough~~, and `inline code`.
```

renders, live:

Bold and **bold**, italic and italic, **bold italic**, ~~strikethrough~~, and `inline code`.

Smart typography

With the typographer on, straight quotes curl, hyphens become dashes, and a few symbols are replaced automatically:

```
"Double" and 'single' quotes, an en -- dash, an em --- dash, an ellipsis...,
plus (c), (r), and (tm).
```

renders, live:

“Double” and ‘single’ quotes, an en – dash, an em — dash, an ellipsis…, plus ©, ®, and ™.

Links

```
An [inline link](https://mantine.dev), a [link with a title](https://mantine.dev "Mantine
docs"),
and a [reference-style link][mantine]. Bare URLs auto-link: https://commonmark.org — as do
angle-bracketed ones like <https://mantine.dev> and emails like <hello@example.com>.

[mantine]: https://mantine.dev
```

renders, live:

An inline link, a link with a title, and a reference-style link. Bare URLs auto-link: https://commonmark.org — as do angle-bracketed ones like https://mantine.dev and emails like hello@example.com.

Inline HTML

Raw HTML passes straight through, which covers the few things CommonMark leaves out:

```
Press <kbd>Cmd</kbd> + <kbd>K</kbd>. You can <mark>highlight</mark> text, write
H<sub>2</sub>O
and E = mc<sup>2</sup>, and define <abbr title="HyperText Markup Language">HTML</abbr>.
```

renders, live:

Press `Cmd + K`. You can highlight text, write H_2O and $E = mc^2$, and define HTML.

A `<details>` element makes a native, no-JavaScript disclosure:

```
<details>
<summary>Show more</summary>

Hidden content that expands when you click the summary.

</details>
```

renders, live:

Show more
Hidden content that expands when you click the summary.

Blockquotes

Aardvark renders a Markdown blockquote as an island rather than a plain `<blockquote>`: a plain `>` becomes an untitled notification — the same one `{% notification %}` emits with its defaults — while an alert marker promotes it to a colored callout (see below). The body is full Markdown:

```
> A blockquote renders as a notification. The body is full Markdown:
>
> - **formatting** and `inline code`
> - lists, too
```

renders, live:

> A blockquote renders as a notification. The body is full Markdown:
>
> - **formatting** and `inline code`
> - lists, too

GitHub-style alerts

Begin a blockquote with an alert marker — `[!NOTE]`, `[!TIP]`, `[!IMPORTANT]`, `[!WARNING]`, or `[!CAUTION]` — and it renders as a callout with a matching color and title:

```
> [!NOTE]
> Useful information that users should know, even when skimming.

> [!TIP]
> A helpful suggestion for doing things better.

> [!IMPORTANT]
> Key information users need in order to succeed.
```

```
> [!WARNING]
> Urgent information that needs immediate attention.

> [!CAUTION]
> Advises about risks or negative outcomes of an action.
```

renders, live:

> Useful information that users should know, even when skimming.

> A helpful suggestion for doing things better.

> Key information users need in order to succeed.

> Urgent information that needs immediate attention.

> Advises about risks or negative outcomes of an action.

Lists

Unordered lists accept `-`, `*`, or `+` as the marker:

```
- First item
- Second item
  - Nested item
  - Another nested item
- Third item
```

renders, live:

- First item
- Second item
- Nested item
- Another nested item
- Third item

Ordered lists use `1.`, `2.`, and so on:

```
1. Step one
2. Step two
3. Step three
```

renders, live:

1. Step one
2. Step two
3. Step three

A numbered list can also **start at a custom number** — the first item sets the starting value and the rest count on from there. Keep it as its own list (separated by intervening text), since adjacent numbered items merge into one list:

```
5. This list starts at five
6. and counts on from there
```

renders, live:

5. This list starts at five
6. and counts on from there

Top-level numbered lists render as a Steps timeline by default — the numbered-badge layout above. Set `steps: false` in `aardvark.config.yaml` to keep plain `<ol>` numbering instead; nested numbered lists and bullet lists are left as-is either way.

Mixed and nested, with a list item that carries several blocks:

```
1. Ordered parent
   - Unordered child
   - Another child
2. Second parent

   A second paragraph that belongs to item 2.

   > And a blockquote, indented to stay inside the item.
```

renders, live:

1. Ordered parent
   - Unordered child
   - Another child
2. Second parent

   A second paragraph that belongs to item 2.

   > And a blockquote, indented to stay inside the item.

Task lists

Begin a list item with `[ ]` or `[x]` and it renders as a Mantine checkbox — checked when the brackets hold an `x` (upper- or lower-case). Everything after the marker is the label, so inline formatting like `code` and **emphasis** works inside it. The boxes are interactive — a reader can toggle them — but the state isn't stored, so it resets on reload to match the source:

```
- [x] Draft the page
- [ ] Review it with the team
- [ ] Ship it, then update `CHANGELOG.md` and **announce**
```

renders, live (try toggling one):

- [x] Draft the page
- [ ] Review it with the team
- [ ] Ship it, then update `CHANGELOG.md` and **announce**

Definition lists (not enabled)

Definition lists are **not** turned on, so the `Term` / `: Definition` syntax falls back to plain text rather than rendering as term/definition pairs:

```
Term
: Definition
```

renders, live (note the leading colon is kept verbatim):

Term : Definition

Tables

Pipe tables come from the enabled `table` rule. A colon in the divider row sets each column's alignment — left (`:---`), center (`:--:`), or right (`---:`). Cells accept inline formatting:

```
| Element      |          Example            |   Done |
| :----------- | :-------------------------: | -----: |
| Bold         |        `**bold**`           |    Yes |
| A link       | [docs](https://mantine.dev) |    Yes |
| Escaped pipe |          a \| b             |    Yes |
```

renders, live:

Element	Example	Done
Bold	`**bold**`	Yes
A link	docs	Yes
Escaped pipe	a | b	Yes

Code

Inline code uses single backticks: `print("hi")`. To include a literal backtick, wrap the span in two backticks: `` `not code` ``.

A fenced block takes an optional language label, and Aardvark highlights it at build time with Pygments — no client-side highlighter or extra JavaScript to load. Under `theme.syntax` in `aardvark.config.yaml` you can pick a preset per color scheme (any Pygments style — the default

`one-light` / `one-dark`, or `monokai`, `dracula`, `nord`, …) and override individual token colors; set `theme.syntax.enabled: false` to turn highlighting off and render a plain `<pre><code>`. Either way the language label survives as a `language-*` class on the `<code>`:

````
```python
def greet(name):
    return f"Hello, {name}!"
```
````

renders, live:

```python
def greet(name):
    return f"Hello, {name}!"
```

```
{
  "name": "aardvark",
  "version": "1.0.0"
}
```

```
vark build && vark dev
```

Indenting a block by four spaces also makes a code block:

```
    This line is indented four spaces,
    so it renders as a code block.
```

renders, live:

```
This line is indented four spaces,
so it renders as a code block.
```

TypeScript & JavaScript: append `twoslash` to a `ts`, `tsx`, `js`, or `jsx` fence to opt that block into **Twoslash** — interactive type hovers, inline compiler errors, and `// ^?` type queries, computed by the real TypeScript language service at build time. See Twoslash.

Code block decorators

A fence's info string (and a couple of inline comment markers) can carry decorators that enrich the rendered block — a title, line numbers, highlighted or focused lines, a diff, soft-wrap, or a collapse toggle. They all render at build time, so a no-JavaScript page shows them correctly; the copy/download buttons and the expand toggle layer on top.

Decorator	Add to the fence	What it does
Title	`title="Any label"`	A header bar with free-form label text — a file path, a description, anything.
Line numbers	`lines`	Numbers every line in a left gutter.

Icon	`icon="key"` or `icon="/logo.svg"`	A Font Awesome glyph (needs `theme.fontawesome`) or an image file, in the header.
Highlight	`{1,3}` or `highlight={1-2,5}`	Tints the listed lines.
Focus	`focus={2,4-5}`	Dims the rest until you hover the block.
Diff	`[!code ++]` / `[!code --]` in a line comment	Marks added / removed lines (any language's comment style).
Wrap	`wrap`	Soft-wraps long lines instead of scrolling.
Expandable	`expandable`	Collapses a tall block behind a Show more toggle.
Local file	`src="path"`	Fills the block from a local file — leave the body empty.
GitHub file	`github="owner/repo/path"`	Fills the block from a public GitHub file — leave the body empty.
Snippet	`snippet="name"`	With `src=`/`github=`, shows only the named Bluehawk region.

The first bare words after the language become the title, so `lines`, `wrap`, and the other flags follow it: a title that needs one of those words should use the quoted `title="…"` form. For example, a title, an icon, line numbers, and a highlighted line together:

````
```javascript title="auth.js" icon="key" lines {2}
export function token(req) {
  const header = req.headers.get("authorization");
  return header?.replace("Bearer ", "");
}
```
````

renders, live:

auth.js

```javascript
export function token(req) {
  const header = req.headers.get("authorization");
  return header?.replace("Bearer ", "");
}
```

The icon is a Font Awesome name or a path to an image file. A bare name is a solid glyph (`icon="key"` → `fa-solid fa-key`) and a brand logo takes the `fa-brands` prefix (`icon="fa-brands fa-square-js"`); a path ending in `.svg`, `.png`, `.webp`, … renders as an `<img>` instead — no Font Awesome required:

aardvark.config.yaml

```
title: My Docs
theme:
  fontawesome: true
```

`focus={…}` dims everything but the listed lines (hover to restore the rest):

```
def total(items):
    return sum(item.price for item in items)
```

`diff` is driven by inline markers — no fence flag needed. Added lines go green, removed red, and the marker comment is stripped from the rendered code and from what the copy button gives you:

````
```js
const config = { debug: false }  // [!code --]
const config = { debug: true }   // [!code ++]
export default config
```
````

renders, live:

```
const config = { debug: false }
const config = { debug: true }
export default config
```

The marker goes in whatever line comment the language uses — `#` in Python, `<!-- -->` in HTML or Markdown, `--` in SQL or Lua. The same change in Python:

````
```python
config = {"debug": False}  # [!code --]
config = {"debug": True}   # [!code ++]
```
````

renders, live:

```
config = {"debug": False}
config = {"debug": True}
```

`wrap` folds a long line onto the next instead of scrolling it off the side — note the absence of a horizontal scrollbar. The `title` here is just a descriptive label, not a file path:

Prompt template

```
prompt = "Summarize the document in three sentences, keeping every date, dollar amount, and proper noun intact, and never inventing facts that are not present in the source text."
```

`expandable` collapses a tall block behind a Show more toggle (shown here with `lines`):

```
def fibonacci(n):
    """Return the first n Fibonacci numbers."""
    seq = [0, 1]
    while len(seq) < n:
        seq.append(seq[-1] + seq[-2])
    return seq[:n]
def is_prime(value):
    if value < 2:
        return False
    for divisor in range(2, int(value ** 0.5) + 1):
```

```
        if value % divisor == 0:
            return False
    return True
def primes_below(limit):
    return [n for n in range(limit) if is_prime(n)]
print(fibonacci(10))
print(primes_below(50))
```

Pulling code from a file

Pasted code drifts — the snippet in your docs and the real, tested file fall out of sync. Instead, let a fence read its body from a file. Leave the body empty (the line after the opening fence is the closing ```) and name a source on the info string.

`src="…"` reads a **local file**, resolved like a local include path: a leading `/` is relative to your content root, any other path is relative to the page (and a read can't escape the project root). The header is titled with the file name automatically (pass `title=` to override), and you can omit the language label to let Aardvark infer one from the file extension. So this fence —

```
```src="/authoring/_examples/greeter.py" lines
```
```

renders the whole file, live — titled `greeter.py`, line-numbered, its language inferred from the `.py` extension:

greeter.py

```
"""A tiny greeting helper — the kind of real, tested source you'd pull a snippet from."""
import os
DEFAULT_NAME = "world"
class Greeter:
    def __init__(self, name: str = DEFAULT_NAME) -> None:
        self.name = name
    # :snippet-start: greet
    def greet(self) -> str:
        """Return a friendly greeting."""
        return f"Hello, {self.name}!"
    # :snippet-end:
if __name__ == "__main__":
    print(Greeter(os.environ.get("GREET_NAME", DEFAULT_NAME)).greet())
```

Showing just a snippet

Those `# :snippet-start:` / `# :snippet-end:` comments in the file above are Bluehawk-style markers. Pull just one marked region with `snippet=`:

```
```python src="/authoring/_examples/greeter.py" snippet="greet" lines
```
```

renders only the `greet` region, live — markers stripped, dedented out of the class body, and clearly marked as a fragment: a **Snippet** badge in the header, faded `⋯` rows top and bottom, and (with `lines`) line numbers that keep the snippet's place in the file rather than restarting at 1:

greeter.pySnippet

```
…
def greet(self) -> str:
    """Return a friendly greeting."""
    return f"Hello, {self.name}!"
…
```

A marker works in any comment style (`#`, `//`, `<!-- -->`, …), may nest, and one name can open several regions (each is dedented, then concatenated). Every other decorator — `title`, `{1,3}`, `focus`, … — still applies, and the copy and download buttons hand back exactly the snippet.

From a GitHub repo

`github="…"` reads a file from a **public GitHub repo** at build time, and renders just like `src=`. A leading `/` uses the repo from your `github.repo` config (on its `branch`); an `owner/repo/path` or a pasted GitHub URL reaches any public repo. Pin a version with `ref=` — a branch, tag, or commit SHA:

````
```python github="/src/app/main.py"
```

```ts github="octocat/Hello-World/index.ts" ref="v2.1.0"
```

```go github="https://github.com/owner/repo/blob/main/cmd/main.go"
```
````

Here's a real one — `yocto-queue`'s `index.js`, pinned to a commit and pulled **live** at build time. The header shows the source repo and filename — each linking to GitHub, next to a GitHub mark — and a footer quotes the repo's license file, so an embedded file always carries its provenance and attribution (it's `expandable`, so it opens on click):

sindresorhus/yocto-queue/index.js

```
/*
How it works:
`this.#head` is an instance of `Node` which keeps track of its current value and nests
another instance of `Node` that keeps the value that comes after it. When a value is
provided to `.enqueue()`, the code needs to iterate through `this.#head`, going deeper and
deeper to find the last value. However, iterating through every single item is slow. This
problem is solved by saving a reference to the last value as `this.#tail` so that it can
reference it to add a new value.
*/
class Node {
    value;
    next;
    constructor(value) {
        this.value = value;
    }
}
export default class Queue {
    #head;
```

```
    #tail;
    #size;
    constructor() {
        this.clear();
    }
    enqueue(value) {
        const node = new Node(value);
        if (this.#head) {
            this.#tail.next = node;
            this.#tail = node;
        } else {
            this.#head = node;
            this.#tail = node;
        }
        this.#size++;
    }
    dequeue() {
        const current = this.#head;
        if (!current) {
            return;
        }
        this.#head = this.#head.next;
        this.#size--;
        // Clean up tail reference when queue becomes empty
        if (!this.#head) {
            this.#tail = undefined;
        }
        return current.value;
    }
    peek() {
        if (!this.#head) {
            return;
        }
        return this.#head.value;
        // TODO: Node.js 18.
        // return this.#head?.value;
    }
    clear() {
        this.#head = undefined;
        this.#tail = undefined;
        this.#size = 0;
    }
    get size() {
        return this.#size;
    }
    * [Symbol.iterator]() {
        let current = this.#head;
        while (current) {
            yield current.value;
            current = current.next;
        }
    }
```

```
    * drain() {
        while (this.#head) {
            yield this.dequeue();
        }
    }
}
```

MIT License

A commit-pinned fetch is cached across builds; a branch or tag is re-read each build so it never goes stale.

It reads the repo's archive, not the GitHub API. Aardvark downloads the repo's zip — the same one `{% gitfolder %}` uses — and reads your file out of it. One download serves every fence and widget naming that repo **at that ref**; since a fence defaults to `main` and the widget to the repo's default branch, name the same `ref` on both if you want them to share one. That means **no GitHub API request, no rate limit, and no token**: nothing here can lose the anonymous 60-requests-per-hour race that a shared CI IP makes easy to lose. The trade is bytes: the whole repo transfers even when you embed one file, so many files from one repo cost one download, while one file from a giant monorepo costs its whole archive. Pinning a commit SHA is the fix worth knowing — it's cached across builds, so it downloads once ever.

That cache is also what keeps a CI job from paying twice: `vark link-check` renders your pages, so it reads the fence's file too. Pinned to a SHA, the check fills the cache and the build that follows reads it; pinned to a branch, nothing is cached across processes and each command downloads the archive once.

A ref counts as an immutable commit — cached across builds — only when it is a full 40- or 64-character hex SHA; don't name a branch as a bare 40/64-hex string, or it would be cached as if it were a commit and never re-read. For a branch or tag whose name contains a slash (`release/2.0`), pass it as `ref=` rather than burying it in a pasted URL, so Aardvark can tell where the branch ends and the file path begins.

The footer quotes the repo's license file. The archive read that fetched your file also finds the repo's root license file and shows the title that file gives itself, linked — the same quote-don't-classify contract the gitfolder footer has, and at the same ref as the code above it. A repo with no license file simply gets no footer.

Public repos only

`github=` reads **public** repos anonymously. It sends no credential, so it cannot read a private repo — and cannot be made to leak one either: an untrusted pull request adding a `github=` fence gets exactly what any anonymous visitor would get. The old `github.allowedOwners` key existed to scope an authenticated fetch; there is no token to scope now, so it is retired and a leftover key warns.

One source per block

A block displays exactly one thing, so naming more than one of an inline body, `src=`, or `github=` **fails the build** — as does a missing file, an unknown snippet name, or a failed read. A broken reference surfaces at build time instead of shipping an empty or stale block.

Diagrams

Label a fenced block `mermaid` and Aardvark renders it to an inline SVG in the browser with Mermaid — flowcharts, sequence diagrams, Gantt charts, and more. The library is fetched only on pages that actually contain a diagram, and each diagram follows the page's light/dark setting:

````
```mermaid
graph TD
  A[Write Markdown] --> B{vark build}
  B --> C[HTML pages]
  B --> D[Search index]
```
````

renders, live:

```mermaid
graph TD
  A[Write Markdown] --> B{vark build}
  B --> C[HTML pages]
  B --> D[Search index]
```

The diagram source stays in the page as plain text inside a `<pre class="mermaid">`, so it still reads sensibly if scripts are disabled. See the Mermaid docs for the syntax of every diagram type.

ASCII diagrams with Markdeep

Label a fenced block `markdeep` to draw **ASCII-art diagrams** with Markdeep: sketch boxes, arrows, and flows in plain text and Aardvark renders them to clean SVG in the browser. Unlike a standalone Markdeep document, you don't add the `***` border — the code fence is the boundary, so just draw the diagram inside it:

````
```markdeep
.----------.   .----------.   .----------.
|  Write   |   |   vark   |   |   HTML   |
| Markdown +-->+  build   +-->+  pages   |
'----------'   '----------'   '----------'
```
````

renders, live:

```markdeep
.----------.   .----------.   .----------.
|  Write   |   |   vark   |   |   HTML   |
| Markdown +-->+  build   +-->+  pages   |
'----------'   '----------'   '----------'
```

As with Mermaid, the Markdeep library is fetched only on pages that contain a diagram, the diagram adapts to the page's light/dark setting, and the raw ASCII stays in the page inside a `<pre class="markdeep">` so it still reads sensibly with scripts disabled. See the Markdeep feature guide for the full diagram syntax.

Horizontal rules

Three or more `-`, `*`, or `_` on their own line render as a Mantine `Divider` — a themed, color-scheme-aware horizontal rule rather than a plain `<hr>`:

```
---
```

renders, live:

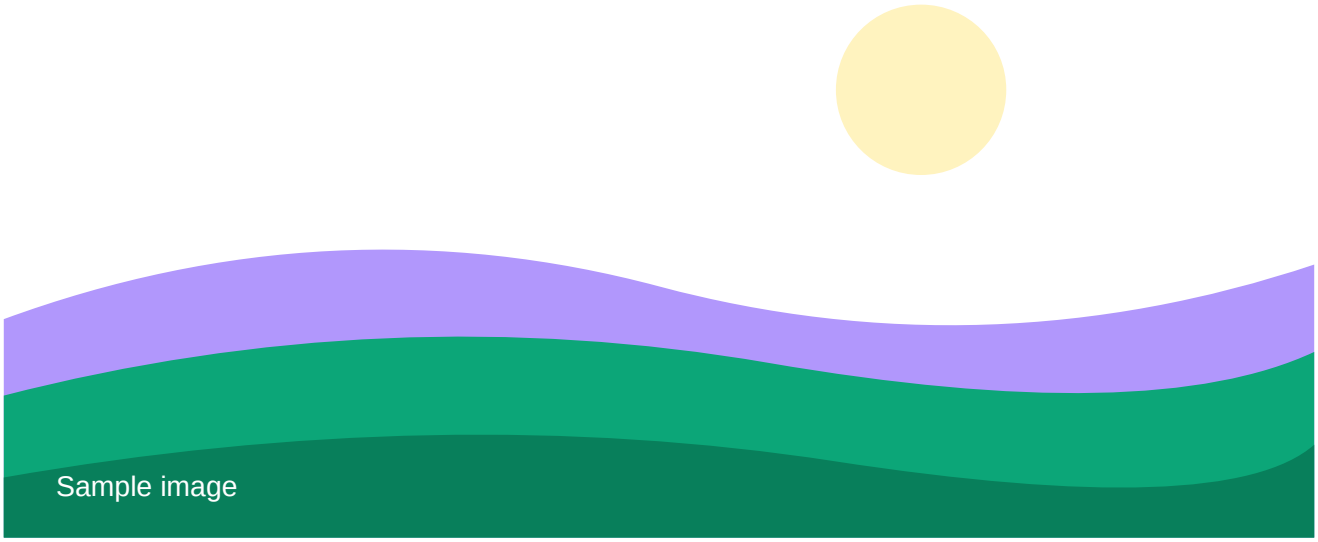
For a **labelled**, dashed/dotted, colored, or **vertical** rule, use the `{% divider %}` tag.

Images

Reference an image with `![alt text](path)`. Files in `static/` are authored from the site root, so `static/img/sample-square.svg` is referenced as `/img/sample-square.svg`. The build emits a fingerprinted file such as `/img/sample-square-<sha>.svg` and rewrites the rendered page. SVG, PNG, and JPG all work the same way:

```
![A sample landscape](/img/sample-landscape.svg)
```

renders, live:



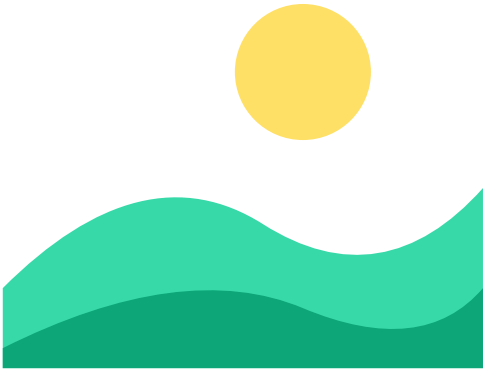
Add a title (shown on hover), make an image into a link by wrapping it, and reuse any static asset such as the site logo:

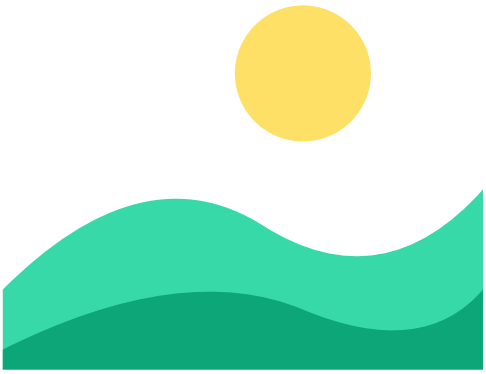
```
![Square thumbnail](/img/sample-square.svg "A square sample image")

[![A clickable thumbnail](/img/sample-square.svg)](https://mantine.dev)

![Aardvark logo](/logo-light.svg)
```

renders, live:





aardvark

Footnotes

Reference a footnote with `[^label]`, then define it anywhere. All footnotes collect at the bottom of the page, each with a link back to where it was used:

```
Here is a numbered footnote[^1] and a named one[^note].

[^1]: The definition can sit right here in the source.
[^note]: Footnotes may contain **formatting** and [links](https://mantine.dev).
```

renders, live:

Here is a numbered footnote[1] and a named one[2].

Escaping and literals

Put a backslash before a Markdown character to render it literally:

```
\*not italic\*, \`not code\`, and 1\. not a list.
```

renders, live:

not italic, `not code`, and 1. not a list.

Aardvark also runs its Python `{% %}` template tags **before** Markdown. A tag wrapped in a raw block prints verbatim — {% badge text='New' %} — while the same tag left unwrapped renders live: [New]. See Templating & data for the full rules on showing literal tag syntax.

Built-in tags

Beyond standard Markdown, Aardvark ships a handful of tags that expand to interactive Mantine components. Each has its own reference page; here is a quick look.

A **badge** for labels, statuses, and counts:

```
{% badge variant='light' color='grape' %}New{% endBadge %}
```

renders, live: [filled] [New] [Beta] [9]

A **callout** is an admonition box, available in four severities:

```
{% callout title="All systems go" severity="success" %}
Your changes were saved.
{% endCallout %}
```

renders, live:

All systems go

Your changes were saved.

Good to know

This setting only applies to new projects.

Double-check this

This overwrites uncommitted local changes.

Destructive action

Deleting a project cannot be undone.

A **notification** is a compact status message with a colored accent line and a Markdown body:

Heads up

The body renders as **Markdown**, including lists and `inline code`.

A **button** renders a link or an action:

Read the quickstart

An **accordion** wraps collapsible panels, each with a full Markdown body:

What is Aardvark?

A Mantine-powered static site generator. This panel body is full **Markdown**.

How do I add a page?

Drop a Markdown file in `content/` with front matter and it joins the navigation automatically.

---

1. The definition can sit right here in the source. ↩
2. Footnotes may contain **formatting** and links. ↩

Twoslash

Twoslash runs the real TypeScript language service over a code block at build time and bakes the results into the page: hover any identifier to see its resolved type, surface expected compiler errors inline, and pin a type with a `// ^?` query. It's the same tooling the TypeScript handbook, VitePress, and Astro Starlight use — rendered with Shiki + `@shikijs/twoslash`.

It's **opt-in per block**: add `twoslash` after the language on a `ts`, `tsx`, `js`, or `jsx` fence. Every other code block is untouched.

````
```ts twoslash
const greeting = "Hello, aardvark"
const loud = greeting.toUpperCase()
```
````

renders, live (hover `greeting` or `loud`):

```ts
const greeting = "Hello, aardvark"
const loud = greeting.toUpperCase()
```

Type queries

A `// ^?` comment, with the `^` under the token you want, pins that token's type below the line — handy for showing what an expression infers to:

```ts
const point = { x: 10, y: 20 }
//    ^?
```

Showing errors

By default Twoslash treats a compiler error as a build problem. To show an error on purpose, declare its code with `// @errors:` — the block then renders with the squiggle and the message inline instead of failing the build:

```ts
// @errors: 2322
const count: number = "not a number"
```

Trimming setup with `// ---cut---`

Real examples often need setup that distracts from the point. Everything above a `// ---cut---` line is compiled (so types still resolve) but hidden from the rendered block:

```ts
// @noErrors
const db = { find: (id: string) => ({ id, name: "Ada" }) }
// ---cut---
const user = db.find("u_1")
```

```
//    ^?
```

Multiple files

Use `// @filename:` to split a block into several modules — imports resolve across them, so you can demonstrate a small project:

```ts
// @filename: shapes.ts
export interface Circle { kind: "circle"; radius: number }
// @filename: area.ts
import type { Circle } from "./shapes"
export const area = (c: Circle) => Math.PI * c.radius ** 2
```

Configuration

Twoslash is on by default. Turn it off site-wide, or pick a different Shiki theme for the rendered blocks, in `aardvark.config.yaml`:

```yaml
twoslash:
  theme: github-light       # Shiki light theme (default: github-light)
  themeDark: github-dark    # Shiki dark theme (default: github-dark)
  timeout: 120              # seconds to allow the render before falling back (default: 120)
```

Set `twoslash: false` to disable the feature everywhere, or add `twoslash: false` to a single page's front matter to disable it just there — useful for a page of deliberately incomplete snippets. Either way, a `twoslash`-tagged fence falls back to a normal highlighted code block.

Requirements

Twoslash needs Node and three build-time packages — `shiki`, `@shikijs/twoslash`, and `typescript` — which ship in the scaffolded `package.json`, so `npm install` once and `vark build` renders your twoslash blocks. If Node or the packages aren't available (or you build with `--no-bundle`), tagged blocks degrade gracefully to plain highlighted code and the build prints a one-line notice — it never fails the build over a missing toolchain.

Keep twoslash snippets short. So the hover and `// ^?` popovers can escape the block, a `twoslash` block isn't horizontally scrollable the way a plain code block is — a very long line extends the page width instead of scrolling. Favor short, focused lines in twoslash blocks.

Build-time Python

Aardvark runs your Python at build time in two places:

- **Inline `{% … %}` blocks** inside any page — to compute part of a page: fetch an API, loop over your `data/`, assemble a table.
- **Generation scripts** in a `generators/` directory — to produce whole files: emit **pages** (one Markdown file per row of data, a stub per API operation) with `generate()`, or emit **any other file** — a downloadable CSV, JSON, or YAML export, an image, a zip, any bytes — with `emit()`.

Reach for an inline block when you want a fragment of an otherwise hand-written page; reach for a generation script when the output is a whole file (or a set of them). The live AI model rates page is a real generation script that writes a page; the downloadable exports at the bottom of this page are a real generation script that writes files. This page you're reading uses an inline block (the live demo below).

Because a generation script can write **any** file the build serves — not just Markdown pages, and not just the formats Aardvark templates into (HTML, `sitemap.xml`, `llms.txt`) — build-time Python is Aardvark's escape hatch for producing whatever a static host can serve: a machine-readable data feed, a `robots`-style text file, a generated diagram, a binary download. If you can compute it in Python, you can publish it.

Inline `{% … %}` blocks

A `{% … %}` block runs Python as the page builds. A single **expression** is rendered in place:

```
This site documents {% len(data.get("products", {}).get("items", [])) %} products.
```

Anything beyond a single expression is **statements** — use the injected `print()` to emit into the page:

```
{%
for item in data["products"]["items"]:
    print(f"- **{item['name']}** — {item['blurb']}\n")
%}
```

`print()` writes its argument **verbatim, with no trailing newline** — so a bare expression never injects stray whitespace. When you emit Markdown line by line (a table, a list), add the `\n` yourself, as above.

Everything in the page's template scope is available inside a block:

Name	What it is
`data`	your parsed `data/` files (the same object templates see)
`site`, `config`, `page`	the site config, the build config, and this page's front matter

`component(...)`, `snippet(...)`	emit any built-in component or project snippet

And because it's just Python, a block can `import` anything and call out to the network.

Use at your own risk

Inline blocks and generation scripts are ordinary, **unsandboxed** Python that runs as part of your build. Only run code you trust. Keep build-time network calls defensive — a timeout and a `try`/`except` fallback — and remember that a block which always hits the network makes **every** build depend on that endpoint being up. `data`, `site`, and `config` are shared by reference across every block and script (and the rest of the build), so **read them, don't mutate them** — a mutation in one leaks into everything that runs after it.

Live demo: this page fetches its own metadata

The viewer below isn't checked in — it's built when this page is. The block fetches `metadata.json` (the agent-discovery artifact Aardvark publishes for this very site) and renders it with the built-in `{% jsontree %}` viewer — the same interactive JSON tree the API reference uses for OpenAPI response examples:

That whole viewer is this block — defensive fetch and all:

```
{%
import json, urllib.request

_url = "https://aardvarkdocs.com/metadata.json"
try:
    _req = urllib.request.Request(_url, headers={"User-Agent": "aardvark-docs"})
    with urllib.request.urlopen(_req, timeout=6) as _resp:
        _meta = _resp.read(4 * 1024 * 1024).decode("utf-8")
    json.loads(_meta)  # validate it parses before handing it to the viewer
    print(component("aardvark", "jsontree", data=_meta, rootName="metadata.json",
                    maxDepth=1, withSearch=True, withExpandAll=True,
                    showItemsCount=True, withKeyCountBadge=True, withBorder=True))
except Exception:
    print("*(The live `metadata.json` couldn't be fetched for this build.)*")
%}
```

Note the shape of a careful build-time fetch: a **timeout**, a **capped read**, a **parse check**, and an `except` that prints a fallback instead of raising. A page block that raises fails the build (by design — a broken page should), so a network call that you don't want to be load-bearing must catch its own failures.

Generation scripts

When the output is a whole file — a page, a page set, or a downloadable data file — move the Python into a `generators/` directory at your project root. A generation script is just build-time Python — the **whole file runs**, top to bottom; there's no entry function to define. Each script gets two writers: `generate()` for **pages** and `emit()` for **any other file**. For pages, on every build Aardvark:

1. **runs** each `generators/*.py` (in alphabetical order) and collects every page it emits;
2. **reconciles** that against `content/`: changed pages are rewritten, pages no longer emitted are deleted, and unchanged pages are left untouched;
3. **discovers and renders** everything in `content/` — generated and hand-authored alike.

So your content tree is always reproducible from `generators/` plus its inputs (stale generated pages can't linger), and a re-run with unchanged inputs writes nothing — which is why `vark dev` can re-run generators on every rebuild without looping on its own output. Files whose name starts with `_` (e.g. `_helpers.py`) are **not** run — they're importable helper modules your scripts can `import`. Skip generators for a build with `vark build --no-generators` (or `vark dev --no-generators`).

Writing pages

A generator gets the same ambient names a page's `{% … %}` blocks get — `data`, `site`, `config` — plus one writer, `generate(path, frontmatter, content)`:

```
import re

# `data` is injected, like in a page. One page per product:
for item in data["products"]["items"]:
    slug = re.sub(r"[^a-z0-9]+", "-", item["name"].lower()).strip("-")
    generate(
        f"catalog/{slug}.md",                                  # path, relative to content/
        {"title": item["name"], "description": item["blurb"]},  # frontmatter
        f"{item['blurb']}\n\n**Price:** ${item['price']}\n",    # Markdown body
    )
```

What's in scope:

Name	What it is
`generate(path, frontmatter, content)`	Write one `.md` page under `content/` — the one guarded writer.
`data`	Your parsed `data/` files (the same object templates see).
`site`, `config`	The site config and build config; `config.root` / `config.content_dir` give you project paths (e.g. for a cache dir).

`generate` is deliberately strict — it's the one supported, guarded way to write:

- the `path` must be **relative** and end in `.md`; a path that escapes `content/` is rejected;
- it **won't overwrite a hand-authored page** — generators only create pages;
- two scripts writing the **same path** in one build is an error, not a silent last-writer-wins.

The `generated` marker

Every page `generate` emits gets two injected front-matter keys:

```
generated: true
generated_by: generators/pricing.py
```

These keys are **reserved**. If your `frontmatter` sets either one yourself, the build **fails** and tells you which key collided — the marker is what lets Aardvark find and wipe generated pages, so a script can't forge or override it.

Because the marker is how Aardvark tells generated pages from hand-authored ones, a file that carries both keys is treated as generator output: on the next build it's overwritten if a generator re-emits it, or deleted if none does. So if you ever want to **keep a generated page permanently as hand-authored** (commit it and edit it by hand), remove both `generated` and `generated_by` from its front matter first — otherwise the next build reclaims it.

Emitting files (CSV, JSON, anything)

`generate()` writes Markdown pages. When you want to publish something that **isn't** a page — a downloadable CSV, a JSON or YAML feed, an SVG, a zip, any bytes — use the other injected writer, `emit()`:

```
import csv, io, json

# One page per row is `generate`; a single downloadable export is `emit`.
# `data` values are read-only DotDicts — project the fields you want into plain
# dicts first, so csv/json/yaml can serialize them.
records = [{"name": r["name"], "price": r["price"]} for r in data["products"]["items"]]

buf = io.StringIO()
w = csv.DictWriter(buf, fieldnames=["name", "price"])
w.writeheader()
w.writerows(records)
emit("downloads/products.csv", buf.getvalue())                  # -> /downloads/products.csv

emit("downloads/products.json", json.dumps(records, indent=2))  # ->
/downloads/products.json
```

`emit(path, content)` writes `content` to `path` in the **build output** and returns the site-absolute URL it will be served at (handy for linking to it from a page you also generate). That's the whole API:

Name	What it is
`emit(path, content)`	Write one arbitrary file to the build output; returns its URL (e.g. `/downloads/products.csv`).

The rules are few and strict:

- **`content` is written verbatim.** A `str` is encoded UTF-8; pass **`bytes`** (or a `bytearray`) for a binary format (an image, a PDF, a Parquet file) and the bytes land untouched — Aardvark never parses, renders, or rewrites an emitted file, so what you emit is exactly what's served.
- **`path` is a clean relative path** under the site root — an absolute path or one containing `..` is rejected. `emit("data/x.json", …)` is served at `/data/x.json`.
- **It won't clobber anything else the build writes** — a page, a `static/` file, or a built-in artifact (`sitemap.xml`, …). Emit to a path nothing else owns.
- **An emitted file is not a page.** It's never discovered, rendered, or listed in the nav, search, or sitemap — it's just a file at a URL. (Link to it with a normal `[download](…)`; link-check skips file links, so a `.csv`/`.json` href is never flagged.)
- **Prefer a file extension.** An extension (`/reports/data.csv`) gives the file its correct content-type everywhere. A suffix-less path (`/reports/data`) is still served — `vark serve` / `vark dev` (and a static host) fall back to the raw file — but with a generic `application/octet-stream` type, so reach for the natural extension when the type matters.

The key difference from `generate()`: a generated **page** is source — it's written into `content/`, then discovered and rendered like hand-authored Markdown, and reconciled there (stale pages are deleted). An emitted **file** is output — it goes straight into the build directory. Because the whole output tree is rebuilt every build, there's no stale-file bookkeeping: a file you stop emitting simply stops appearing.

No new formats to learn

This is why Aardvark doesn't need Hugo-style "output formats" or per-page alternate templates. Anything a static host can serve, a generator can write with a few lines of ordinary Python — `csv`, `json`, `yaml`, `PIL`, whatever's on PyPI — reading from the same `data/` your pages use.

A live example

This very site's **AI model rates** page is built by one generation script: `generators/pricing.py` calls **OpenRouter's API** at build time and writes `pricing/models.md` — a real-world generator that fetches live data (with a cached fallback under `config.root / ".aardvark-cache"`) rather than reading a local file.

The result is an ordinary page: it appears in the nav, in search, in the sitemap, and is **link-checked** like everything else — a generated page with a dead link fails the build, exactly as a hand-authored one would.

And the `emit()` side is live too: `generators/downloads.py` reads the same `data/products.yaml` this site's shop demo uses and publishes the catalog as three real downloads — built fresh on every build, byte-for-byte from the data:

- products.csv
- products.json
- products.yaml

None of those are pages — they're files at a URL, produced by ~20 lines of ordinary Python.

Security & anti-patterns

Build-time Python is **trusted-author code, not a sandbox**. A generator or an inline block runs with your build's full privileges — filesystem, environment variables, network — the same as a `Makefile`, a Sphinx `conf.py`, or an npm `postinstall` script. There's no in-process restriction, and there can't be a meaningful one (hiding `os` wouldn't stop a determined script and would break legitimate uses like authenticated fetches). So the rules below are about who runs what, not about locking the language down.

This matters most for public repos

On a **private** repo — or any repo where only people you trust can push and open pull requests — build-time Python only ever runs your own code: review it like any other code and you're covered.

Avoid:

- **Running generators or inline blocks you don't trust.** They can read your environment variables, read and write files, and reach the network. Treat a change that adds or edits a generator (or a build-time block) as a code change with that power, and review it as one.

Letting untrusted code run with secrets in the environment. The defense is configuration you click through in your repo's GitHub settings:

- **Settings → Actions → General → "Fork pull request workflows from outside collaborators" → "Require approval for all outside collaborators."** A stranger's PR then runs no workflow until you press **Approve**, so their code never runs unreviewed.
- **Settings → Environments** → keep deploy secrets in an environment whose **Deployment branches** are limited to your release branch (not in loose repository secrets). Only the post-merge deploy can read them — a PR build can't.
- **Settings → Actions → Runners** → don't attach self-hosted runners to a public repo; a fork PR's code would run on your machine.
- One gotcha no toggle covers: a workflow with **`on: pull_request_target`** runs with your secrets even for fork PRs — if you have one, make sure it never checks out and runs the PR's code. Plain `on: pull_request` is the safe default (fork PRs get no secrets).

- **Building genuinely untrusted content in your normal build environment.** If you must build content you don't trust, isolate the whole build in an ephemeral container with **no secrets in its environment** — the settings above keep your secrets out of an outside PR's reach, but only

isolation contains code you can't review at all.

- **Hardcoding secrets in a generator.** Read them from the environment (`os.environ`) — fine for your own build — and never commit the value.
- **Unguarded build-time network calls.** A bare `urlopen` makes every build depend on a third party being up. Wrap it in a timeout + `try`/`except` with a fallback, and cache when you can — as `generators/pricing.py` does.
- **Mutating `data`, `site`, or `config`.** They're shared by reference across every block and script in the build — read them, don't mutate them, or you'll poison whatever runs next.

Components & snippets

Call `component('Name', **props)` to drop a React component into a page. At build time Aardvark records a mount point and bundles the component; in the browser it mounts as an **island** wrapped in a Mantine provider.

> **How islands render:** components are **client-rendered** — the bundled runtime mounts each one with React (`createRoot`) on load. Enabling `islands: { ssr: true }` additionally prerenders the widgets to static HTML at build time, so crawlers and no-JS readers see real markup; the client still re-renders on load (it does **not** hydrate / attach to that markup, so there's no server/client mismatch to worry about). This isn't Astro-style partial hydration — the whole component bundle ships and every island re-renders.

Mantine components

Any Mantine component is available by name:

```
{% component('Button', children='Get started', color='blue') %}
```

renders, live:

Get started

- `children` is the content (text, or another `component(...)` call).
- All other keyword args become props. They must be JSON-serializable, so pass data and `defaultValue`/`defaultChecked` rather than functions. To attach an event handler or any raw HTML attribute, use the `attr` argument (below).

See the Component gallery for live examples.

Custom snippets

Put a React component in `snippets/` and reference it by filename. This site includes `snippets/ProductCard.jsx`:

```
{% component('ProductCard', product='Aardvark Pro', tagline='Docs that build themselves',
badge='New') %}
```

`snippet(...)` is an alias of `component(...)` if you prefer to make the intent explicit.

Overriding a built-in

A snippet **wins any name collision** with a Mantine component or a built-in island. Name a file after one — `snippets/Button.jsx`, `snippets/Survey.jsx` — and `component('Button')` mounts your component everywhere, not the shipped one; Aardvark bundles your file in its place. That's the supported way to customize a built-in island wholesale (restyle it, swap its data source, change its behavior) without forking the theme — for example, replacing the built-in survey card with your own `snippets/Survey.jsx`. The precedence is Mantine → built-in island → your snippet, last one wins, so your `snippets/` always takes priority.

A snippet is also a tag

Putting a file in `snippets/` gives you a matching `{% Name %}` **directive** too — the same way a custom component does. Write it as a tag, pass scalar props as attributes, and let the body become the snippet's `children`:

```
{% ProductCard product="Aardvark Pro" tagline="Docs that build themselves" badge="New" %}
```

That's exactly `{% component('ProductCard', …) %}`, so reach for whichever reads better. Two things only the call form can do: pass a **non-scalar prop** (a list or dict — tag attributes are scalar, like every other directive), and run inside a `for` loop (a `{% Name %}` tag is expanded when the page is scanned, before any loop runs). The `attr={…}` escape hatch below works in both forms. See Custom snippets for the full story and when to choose a snippet over a custom component.

Attaching HTML attributes (attr)

Components render to real DOM in the browser, so you never write their final tag — which means there's normally nowhere to hang an `onclick`, a `data-*` hook, or an `id`. The `attr` argument fills that gap: pass a dict of raw HTML attributes and they're applied to the component's **rendered root element** when it mounts.

```
{% component('Button', children='Click me', attr={'onclick': "this.textContent='Clicked!'",
'data-track': 'cta'}) %}
```

renders a real button whose `onclick` runs your JavaScript — click it:

Click me

Because the value is just a string of JavaScript — and nothing says it has to be a single line — you can drop a whole anonymous function straight into `attr`. A triple-quoted Python string carries a multiline body, so the handler can do real work inline (here an IIFE; swap in a `fetch`, analytics, a modal — anything):

```
{% component('Button', children='Say hello', attr={'onclick': '''
  (() => {
    const who = 'world';
    alert(`Hello, ${who}!`);   // …or call any API you like
  })()
'''}) %}
```

Say hello

Values may be strings, numbers, or booleans (`True` → a bare boolean attribute; `False`/`None` → omitted). They're applied with `setAttribute`, so a value can only set an attribute on that one element — it can never inject markup elsewhere.

- Use `attr` for **event handlers** (`onclick`, …), **`data-*`** / **`aria-*`**, `id`, and other custom attributes.
- Use normal **props** for `className` and `style` — React manages those, and an `attr` would fight it (Aardvark warns in the console if you try).
- A few Mantine components render only context and have no DOM node of their own — `Accordion`, `Combobox`, `MenuSub`, `NumberFormatter`. `attr` on these applies only at the top level; nested, put the `attr` on a child element instead. (`Menu`, `Popover`, and `HoverCard` do take `attr`, but it lands on the rendered `.Dropdown` overlay — not the trigger.)
- Custom snippets support `attr` when they **forward their ref** to a root element — see `snippets/ProductCard.jsx`. Every Mantine component supports it out of the box.

A **custom class or style** is the one case to reach past `attr`: pass them as the `className` and `style` props so Mantine merges them with its own. (Setting `class` or `style` through `attr` would overwrite the component's — Aardvark warns if you try.)

```
{% component('Button', children='Custom look', className='cta-pill', style={'borderRadius':
'999px', 'textTransform': 'uppercase'}) %}
```

Custom look

`cta-pill` lands on the rendered button alongside Mantine's own classes (ready for your CSS to target), and the inline `style` is merged in — neither clobbers the component's styling.

`attr` is intentionally powerful (it can run JavaScript), in keeping with Aardvark's trusted, in-repo content model. A site that wants to restrict it can set an `attrPolicy` in `aardvark.config.yaml` — patterns may end in `*` for a prefix match:

```
attrPolicy:
  deny: ['on*']                 # block inline event handlers
  # allow: ['data-*', 'aria-*', 'id']   # …or allowlist-only
```

Nesting

Because a `component(...)` call returns its mount markup, you can nest calls by passing one as another's `children`:

```
{% component('Group', children=component('Badge', children='New') + component('Badge',
children='Beta', color='grape')) %}
```

[New][Beta]

This is how compound components (Accordion, Tabs, Card sections, …) are built — see the gallery for examples. For a compound whose body is **Markdown** rather than a string, reach for a block component instead.

To **reuse** a chain like this — name it, give it typed parameters, and call it as `{% MyTag … %}` instead of repeating the `component(...)` calls — define a custom component. It's the recommended way to package a composition for reuse.

Custom components

Chaining `component(...)` calls inline is fine once, but it isn't reusable — the composition can't be named, shared, or given a typed interface. A **custom component** fixes that: define it once in a `.md` file under `components/`, with front matter declaring its **tag name** and **parameters** (each with a type, an optional default, and whether it's required), and a `{% %}` template body holding the `component(...)` calls. Then use it by its tag.

A custom component is a **build-time macro**: at build it expands into the components it wraps and ships **no** JavaScript of its own. (That's the difference from a `snippets/*.jsx` React component — see the bottom of this page.)

Define one

`components/BadgeGroup.md` turns the classic "Group of Badges" chain into a tag:

```
---
name: BadgeGroup          # REQUIRED — the tag you'll write as {% BadgeGroup %}
params:
  gap:
    type: string
    default: xs
---
{% component('Group', gap=gap, children=children) %}
```

The declared params (`gap`) and the special `children` slot are available as variables in the body.

Use it — inline or block

Self-closing when there's no body, or a paired tag (`{% end<Tag> %}`, first letter capitalized) when you want to wrap content. The wrapped content renders in the page and arrives as `children`:

```
{% BadgeGroup %}{% component('Badge', children='New') %}{% component('Badge',
children='Beta', color='grape') %}{% endBadgeGroup %}
```

renders, live:

[New][Beta]

`children` holds whatever you wrapped — inline text, `component(...)` calls, even other custom components. If you wrap **Markdown** (headings, lists, prose), pad the slot with blank lines in your definition so the page's Markdown pass renders it: `{% component('Card', children="\n\n" + children + "\n\n") %}` (the same trick block components use).

If a macro needs audience-specific `changelog` or `openapi` output, put that directive directly inside a paired `{% visibility %}` block in the macro. Do not put visibility around `{% children %}` at that point: the children have already rendered their page-level RSS/navigation side effects, so Aardvark fails the build rather than publish them to the wrong audience.

Parameters

Each entry under `params:` declares a typed input:

```
params:
  title:
    type: string
    required: true        # no default → must be supplied
  color:
    type: string
    default: blue         # supplied or this
  size: int               # shorthand: a bare type = optional, no default
```

- **Types**: `string`, `int`, `float`, `bool`, `list`. Values from the call site are coerced to the declared type (so `count="3"` becomes the integer `3`, and a `list` accepts a comma-separated string). A value that can't be coerced is a build error.
- **`required: true`** with no value supplied is a build error; **`default`** fills in when the param is omitted. Declaring both is rejected (a default already makes it optional).
- Passing a param the component didn't declare is a build error — every mistake is caught at build time, naming the component, the param, and the file.

The calling page is **not** visible inside the body — a component is a pure function of its declared inputs. Pass page data explicitly, e.g. `{% Hero title=page.title %}`. For the same reason, emit from a body with `print(...)` or `{% value %}` — `page.print()` isn't available here, since there's no `page` in scope.

A definition body can blend anything

The body is a full template, so it can freely mix **Mantine `component()` calls**, real **Python** (`{% %}` blocks), raw **HTML/CSS**, **`<script>`** tags, and **`attr={...}`** event handlers. `components/CopyCard.md` does all of it — a Python expression, a Mantine `Button`, a `<style>` block, an `onclick` + `data-*` via `attr`, and a top-level `<script>`:

```
{% CopyCard label='Copy install command' text='pip install aardvark' %}
```

renders a working copy button — click it:

Copy install command

20 characters · click to copy

Two things worth knowing about injected JavaScript:

- A **`<script>` at the body's top level** runs on page load. A `<script>` placed inside a `component(...)` call becomes a React child and is rendered inert (browsers don't execute scripts inserted that way) — keep runnable scripts at the top level.
- Use **`attr`** for per-component handlers/data (`onclick`, `data-*`, `id`); use the `className`/`style` **props** for styling (React owns those). A site can restrict `attr` with `attrPolicy` in `aardvark.config.yaml`.

Built-in components

Aardvark ships a few **built-in components** so you don't have to define them. `{% button %}` renders a button or link, exposing the full Mantine Button surface — `variant`, `color`, `size`, gradient, sections, link target, spacing, `id`, and more. See its page, Button, for the full list with live examples. The label is the block body or a `text` param.

```
{% button text='Get started' url='/start/' color='grape' %} {% button url='/docs/'
variant='outline' %}Read the docs{% endButton %}
```

renders, live:

Get started

Read the docs

The header's top-bar buttons (`topButtons` in `aardvark.config.yaml`) use this same component.

Another built-in is `{% callout %}` — an admonition: set `title` for an optional bold heading and `severity` (`success` / `info` / `caution` / `warning`) to color the box and pick its icon. See its page, Callout, for parameters and examples.

To customize a built-in, define your own `components/<name>.md` with the same `name:` — your version wins.

Aardvark also ships `{% callout %}` — a titled, colored admonition box. Set `severity` to one of `success` (green), `info` (primary), `caution` (yellow), or `warning` (red); `title` is optional; the block body is the message. Close it with `{% endCallout %}`.

```
{% callout title="This is a destructive action" severity="warning" %}
Be careful when proceeding here.
{% endCallout %}
```

renders, live:

This is a destructive action

Be careful when proceeding here.

`callout` pairs its `.md` definition with a built-in React snippet (`Callout.jsx`) that maps each severity to its color and icon — a small example of a built-in component composing a snippet.

Custom components vs snippets

Both let you make your own building blocks; reach for the right one:

- **Custom component** (`components/*.md`) — composes existing components (Mantine, builtins, snippets) into a named, parameterized tag. No JavaScript is added. This is the recommended path for reuse, and what you want most of the time.
- **Snippet** (`snippets/*.jsx`) — a genuinely new **React component** written in JSX, for behavior/markup that composition can't express. It's bundled and mounted client-side, and earns

its own `{% Name %}` tag too. See Custom snippets for how to write one and a full side-by-side breakdown.

Rule of thumb: composing what already exists → custom component; writing new React → snippet.

Custom snippets

A **custom snippet** is a real React component you drop in `snippets/*.jsx`. Reach for one when you need actual React — state, effects, hooks, a third-party React library, or a browser API — that you can't express by composing existing components. Unlike a custom component, which is a build-time macro that ships **no** JavaScript, a snippet **is** JavaScript: it's bundled and mounts as an island in the browser.

Write one

A snippet is a `.jsx` file under `snippets/` that default-exports a React component. Its props arrive as the component's props, and anything you wrap becomes `children`:

```
// snippets/Stepper.jsx
import { useState } from 'react';
import { Button } from '@mantine/core';

export default function Stepper({ label = 'Clicks', start = 0 }) {
  const [n, setN] = useState(start);
  return <Button onClick={() => setN((c) => c + 1)}>{label}: {n}</Button>;
}
```

That `useState` is the tell — it holds live client state, which a `.md` macro can't. The file name (`Stepper`) is the component name.

Two ways to use it

A snippet is reachable both as a **call** and as a **tag**; both mount the same island, so use whichever reads better. This site ships `snippets/ProductCard.jsx`:

```
{% component('ProductCard', product='Aardvark Pro', badge='New') %}   {# call form #}
{% ProductCard product="Aardvark Pro" badge="New" %}                  {# tag  form #}
```

both render the same island, live:

- **Tag form** — `{% ProductCard … %}`. Defining the snippet gives you the tag, the same way a custom component does. Scalar attributes become props; a wrapped body (closed with `{% endProductCard %}`, first letter capitalized) becomes `children`. The `attr={…}` escape hatch from Components & snippets works here too.
- **Call form** — `{% component('ProductCard', …) %}`. Needed for the two things a tag can't do: pass a **non-scalar prop** (a list or dict — tag attributes are scalar, like every directive), and emit from inside a **`for` loop** (a `{% Name %}` tag is expanded when the page is scanned, before any loop runs, so loops use the call form — see the storefront example).

Snippet or custom component?

Both let you define your own building block and call it as a `{% Tag %}`. The difference is **what's behind the tag**:

	Custom component (`.md`)	Custom snippet (`.jsx`)
What it is	A build-time **macro**	A real **React component**
Ships JavaScript?	No — expands into `component()` calls at build	Yes — bundled, mounts as an island
Body written in	A `{% %}` template	JSX
Reach for it when	**Composing existing** components into a named, parameterized tag	You need **new React**: state, effects, hooks, a third-party React lib, browser APIs
Tag form	`{% Name %}`	`{% Name %}`
Call form	—	`{% component('Name', …) %}`

Rule of thumb: composing what already exists → custom component; writing new React → snippet. Both earn a `{% Name %}` tag, so the call site reads the same — the choice is only about whether you're wiring existing pieces together or writing genuine React.

A case that has to be a snippet

The Stripe provider on the Component libraries page is the canonical "must be a snippet." Mounting Stripe's card fields needs `loadStripe('pk_…')` — which returns a Promise and injects Stripe.js — wrapped in Stripe's `<Elements>` React context provider. None of that is expressible as a build-time macro (there's no JavaScript to run, no React context to hold), so `StripeProvider` is a `snippets/*.jsx` file. Because it's a snippet, you still get the natural `{% StripeProvider %}` tag — that's exactly this behavior at work.

Block components

`component('Name', …)` drops a single component inline, and its `children` is a plain string — Markdown inside it is **not** rendered. **Block components** solve the other case: a compound built from several Mantine components, given its own tag pair, wrapping a region of **Markdown that renders normally**.

The first one is the **accordion**:

```
{% accordion %}
{% accordionSection title="Section One" %}
## Markdown content

**This builds great!**
{% endAccordionSection %}
{% endAccordion %}
```

renders, live:

Section One

Markdown content

This builds great!

Section Two

More Markdown content

- This bulleted list actually renders!
- See?

Each section's `title` is the clickable control, and everything between the tags is its body — full Markdown (headings, lists, code, links), plus any `component(...)` call or even a nested block.

How it works

At build time the `{% %}` engine expands the block into nested Mantine islands (`Accordion` › `Accordion.Item` › `Accordion.Control` + `Accordion.Panel`) and lets the page's single Markdown pass render each panel body. In the browser the island mounts like any other component.

Built-in block components

Each has its own reference page under **Built-in Components**:

- Accordion — collapsible sections with Markdown bodies.

- Article card — blog-style article cards with a byline, avatar (or initials fallback), date, badge, and cover image, in five layout variants (`{% articleCard %}`).
- API fields — hand-authored parameter and response-field rows for an API reference that isn't driven by an OpenAPI spec: a field's name, type, required/deprecated badges, an optional default and request-location, beside a Markdown description (`{% field %}`).
- Card — content cards with icons, cover/background images, gradient/glass/stat variants, and whole-card links, arranged in a responsive grid (`{% card %}` / `{% cardGrid %}`).
- Code groups — several fenced code blocks shown as language/file tabs, each with its own copy button (`{% codeGroup %}`).
- Examples — side-by-side request/response panels for API docs: titled, syntax-highlighted code with per-block copy/download and a tab strip (`{% requestExample %}` / `{% responseExample %}`).
- Include — splice a shared Markdown partial into a page, varied by the page's front matter.
- Map — an embedded OpenFreeMap / MapLibre map with a pin per location, placed by address (geocoded at build time) or by coordinates.
- Panel — a supplementary side panel that floats beside the content on a wide viewport and stacks below it on a narrow one (`{% panel %}`).
- Prompt — a copyable AI-prompt block with verbatim prompt text plus copy and "Open in ChatGPT / Claude / Cursor" actions (`{% prompt %}`).
- Tabs — tabbed panels with full Markdown bodies (`{% tabs %}` / `{% tab %}`).
- Tree — a nested file/folder explorer (`{% tree %}` / `{% folder %}` / `{% file %}`); click a file to open its source in a modal.
- Update — an inline release-note entry on a timeline rail: a version/date label beside its Markdown body (`{% update %}`).
- Visibility — show or hide a block depending on whether a human is reading the HTML page or an AI agent is reading its Markdown twin (`{% visibility %}`).

Defining a new block component is a small addition to the build tool — see `AGENTS.md`.

Custom CSS & JS

Root stylesheets and scripts

Any `.css` or `.js` file in your project root is copied into the build, given a per-build fingerprint, and linked automatically — stylesheets in `<head>`, scripts (deferred) before `</body>`:

```
my-docs/
  custom.css     ->  /custom-<sha>.css, linked on every page
  analytics.js   ->  /analytics-<sha>.js, loaded on every page
```

No configuration needed. The scaffold includes a `custom.css` you can edit.

Static files

Anything under `static/` or `public/` is copied into the build root, preserving the authored path but emitting cacheable files with a per-build fingerprint:

```
my-docs/
  static/
    img/logo.svg   ->  /img/logo-<sha>.svg
    fonts/ui.woff2 ->  /fonts/ui-<sha>.woff2
```

Reference them with their stable authored paths (`/img/logo.svg`); Aardvark rewrites generated HTML, CSS, JS, manifests, and component props to the fingerprinted output URL. For dynamic template values, use `{% asset('/img/logo.svg') %}`.

Fingerprinted output

Authors write stable URLs; builds publish fingerprinted asset URLs. For example, keep writing `/img/logo.png` in Markdown, HTML, CSS, JS strings, or component props. `vark build` rewrites those references to the emitted file, such as `/img/logo-<sha>.png`, preserving any query string or fragment.

The token is one value shared across the whole build — the current short git `HEAD` SHA. When the build can't read git (for example a container build without the `.git` directory), Aardvark uses the commit SHA from the `AARDVARK_BUILD_SHA` environment variable if you set it, otherwise a single content hash of your project sources. Because every asset in a build shares one token, a reference and the file it points at always rotate together, so an aggressively cached stylesheet can never end up pointing at an image URL a later build renamed away. Fingerprinted files are emitted only at their fingerprinted paths — there is no legacy `/img/logo.png` copy and no redirect — so your static host or CDN can cache those URLs aggressively.

This also applies to `.txt` and `.json` files you place in `static/` or `public/`. If another site links directly to `/release-notes.txt` or `/feed.json`, that stable URL will not exist after the build; keep externally linked documents as authored pages or generated discovery surfaces when the URL itself needs to stay permanent.

Route and discovery files stay stable instead of fingerprinted: page HTML, `_headers`, `_redirects`, `robots.txt`, `sitemap.xml`, `.well-known/**`, `auth.md`, `llms*.txt`, `metadata.json`, `search-index.json` (and `.gz`), page `.md` siblings, encrypted `.enc` payloads, and internal config documents. Serve those with normal revalidation so readers and agents see fresh metadata after a deploy.

Theme assets

The theme's own CSS/JS live in `themes/vark/` and are emitted under `/_aardvark/` with the same fingerprinting (for example, `/_aardvark/theme-<sha>.css`). See Theme & customization to change them.

Navigation menus

Navigation has two parts: the **horizontal tabs** (in `aardvark.config.yaml`) and each tab's **left-nav menu** (defined in page front matter). The config stays tiny — no hand-maintained nav tree — and the sidebar is a property of your content. The tabs are optional: omit them and you get just the sidebar, or turn a tab into a dropdown when it fronts a deep section.

Tabs (config)

Each tab gets a `link` and an `id`. The `id` is also the **menu name** that pages target. Add `icon:` to show a glyph beside the tab label.

```yaml
tabs:
  - label: Docs
    icon: book-2
    link: /
    id: docs
  - label: Components
    icon: components
    link: /components/
    id: components
```

A tab is highlighted whenever the current page belongs to its menu, so a tab linked to `/` stays active on its nested sub-pages.

Dropdown tabs

Give a tab `items:` instead of a `link:` and it becomes a **dropdown** — clicking the tab opens a menu of child links (hover and keyboard both work, with a no-JS fallback). Each child takes a `label`, a `link`, and an optional `icon:`. This is handy when one tab fronts a deep section and you want to expose its sub-areas up top. Keep the tab's `id:` — it still names the left-nav menu, so your `menu:` front matter is unchanged and the tab stays lit across the **whole** section, not only the sub-pages it lists:

```yaml
tabs:
  - label: Docs
    icon: book-2
    id: docs            # names the "docs" menu; no `link:`, so this tab is a dropdown
    items:
      - label: Getting Started
        icon: flag
        link: /docs/
      - label: Authoring
        icon: pencil
        link: /authoring/templating/
      - label: Theming
        icon: palette
        link: /theming/
```

```
      - label: Deploy
        icon: rocket
        link: /deploy/
```

Because the dropdown children are ordinary links, don't give them their own `id:` unless you mean to open a **separate** menu — an `id` on a child names its own menu the same way a top-level tab's does. Point children at pages that already live in the section (the `menu: docs` pages, here) so their landing links stay in sync.

Optional: a site with no tabs

The horizontal tab bar is optional. **Omit `tabs:` entirely** and no bar renders at all — every page just shows the left-nav sidebar for its own menu. Give a section's `index.md` a `menu:` and the rest of the folder auto-joins it; the menu name no longer has to match a tab.

```
# aardvark.config.yaml — no `tabs:` key at all
site:
  name: My Docs

---
# content/guide/index.md
title: Guide
menu: guide     # names this section's sidebar; siblings auto-join, no tab needed
---
```

Reach for this when a site is a single flat section (or a handful you'd rather not surface as tabs) — you keep the content-driven sidebar with zero header chrome.

Menus (front matter)

Add a page to a tab's left nav with front-matter keys:

Key	Meaning
`menu`	The tab `id` this page joins (top-level placement).
`nav`	`nav: false` keeps the page **in** its menu (so its tab stays lit when you visit it) but **out** of the sidebar list — for pages readers reach through a listing rather than the nav, like this site's knowledge-base articles and changelog entries. (Whether to hide a taxonomy's members is per-site taste: our blog keeps its posts in the sidebar as an archive.)
`parent`	The `id` of the parent entry (nests this page under it).
`id`	This entry's id, for children to reference. Defaults to the page's path.

`navtitle`	A shorter label for the sidebar and breadcrumbs when the page `title` is too long for the narrow nav. The `<title>` tag and on-page H1 keep the full `title`.
`weight`	Optional ordering override. Siblings list **alphabetically by label** by default; give a page a `weight` to pull it ahead (lowest first).
`heading`	A label string rendered above this entry as a section divider; the page itself stays a normal, clickable entry.
`icon`	An icon shown to the **left** of this entry's label (see Icons).
`heading-icon`	An icon for the `heading` section label above this entry.

Pages can also carry `taxonomy:` front matter — tag-based membership that drives changelog / knowledge-base / blog listings and can add a per-page tag nav to the sidebar; see the Taxonomy component.

Entries list **alphabetically by their nav label** out of the box, so new pages slot into place with no bookkeeping. Reach for `weight` only when you want a specific order — a weighted entry jumps ahead of the alphabetical ones. And when a page's `title` is too long for the narrow sidebar, give it a short `navtitle`:

```
---
title: Templating & data
navtitle: Templating # short sidebar label; <title> and H1 keep "Templating & data"
menu: docs           # joins the "docs" tab's left nav
parent: authoring    # nested under the entry whose id is "authoring"
weight: 10           # optional — overrides the alphabetical default
---
```

Sections from folders

The easy path: give a folder's `index.md` a `menu` (or `parent`), and **every other file in that folder joins it automatically** — no per-file front matter. A page's id defaults to its path, so `components/button.md` has id `components/button` and `modes/index.md` has id `modes`.

```
content/components/
  index.md        # menu: components  -> the "Components" section (a real page)
  button.md       # auto-joins Components
  action-icon.md  # auto-joins Components
```

Two kinds of entry, plus labels

- **Leaf** — a page with no children: a plain link.

- **Branch** — a page that has children (e.g. a folder `index.md`): its label links to its own page, and a **caret** expands/collapses the subtree. Click the caret to peek inside without navigating.

A `heading:` string adds a **non-clickable section label** above an entry — the "Getting Started" and "Authoring" labels in this sidebar are leaves whose first item carries a `heading`, with the rest of the group as plain siblings beneath it. The current page's ancestor branches render expanded, so the nav is correct even before JavaScript loads.

Icons

Use `icon:` on horizontal tabs, dropdown tab children, or pages in the left nav. The glyph renders to the **left** of that item's label. `heading-icon:` does the same for the `heading` section label above a left-nav entry — the "Getting Started" group in this sidebar shows both. A value is read **exactly like an `{% icon %}` spec** — same sources, same rules:

- **A Tabler icon name** — a bare lowercase name like `rocket` or `brand-github` (the default for a bare name). Aardvark bakes Tabler glyphs directly into the static navigation — from the outline set bundled with Aardvark, or your locally installed `@tabler/icons` package when you've pinned a different `theme.iconVersion` — so they paint with the page and tint with the link color, with no JavaScript or icon request on the normal path.
- **A Font Awesome class** — explicit, e.g. `fa-solid fa-rocket`, `fab fa-github`.
- **A path to an image** — any `.svg`/`.png`/`.jpg`/… under `static/`, e.g. `/icons/folder.svg`. Square images read best.
- **An emoji** — any non-ASCII glyph, e.g. `🚀`.

The icon is capped to one line so it never makes the nav row taller.

```
---
title: Quickstart
menu: docs
icon: bolt                      # a Tabler icon, left of this page's label
heading: Getting Started
heading-icon: fa-solid fa-flag # a Font Awesome glyph, left of the section label above it
---
```

A bare name is a **Tabler** icon — Font Awesome must be written out (`fa-solid fa-bolt`), exactly as in the `{% icon %}` tag. Font Awesome glyphs also need the FA stylesheet loaded: set `theme.fontawesome: true` in `aardvark.config.yaml` (this site does), or pass a kit/CDN URL to self-host or use Pro. Tabler, image, and emoji icons need none of that. A custom `theme.iconCdn` is authoritative (it may serve different SVGs): navigation uses its runtime loader and `vark build` stays quiet. A `theme.iconVersion` that is neither bundled with Aardvark nor installed locally also falls back to the runtime loader — but there `vark build` warns which glyphs it couldn't bake.

Tables

Write a normal GitHub-flavored Markdown table and the default theme enhances it automatically — there's no special syntax or component to reach for:

- **Sortable headers.** Each column header is a button: click it to sort by that column, click again to reverse. Columns whose cells are all numbers sort numerically (commas, a leading currency sign and a trailing `%` are tolerated); everything else sorts as text.
- **Filter box.** Hover a table — or tab into it — and a search field appears above it; typing filters the rows live.
- **Styling.** The header row is filled with the body text color (with the page background as its text), and rows are lightly striped for readability. Both track the active light/dark color scheme.

It's progressive enhancement: with JavaScript disabled the table still renders styled and striped — only the sorting and filtering need the small, dependency-free `tables.js` the theme ships.

Example

Click **Monthly downloads** to sort by size, or hover the table and type to filter.

Package	Version	Monthly downloads
markdown-it-py	3.0	1,200,000
mantine	9.3	890,000
esbuild	0.21	4,500,000
click	8.1	9,800,000

Turning sorting or filtering off

Both behaviors are on by default. Turn either off for the **whole site** in `aardvark.config.yaml`:

```yaml
tables:
  sortable: false      # no click-to-sort headers
  filterable: false    # no filter box
```

A bare `tables: false` turns both off. Override per **page** in its front matter — a page setting always wins over the site config:

```yaml
---
title: Release notes
sortableTables: false
filterableTables: false
---
```

The `sortable` and `filterable` switches also govern the tables in an API reference rendered with the `{% openapi %}` directive. Either way the table stays styled, striped and horizontally scrollable — only the interactivity is removed.

Notes

- Enhancement applies to any table with a header row and at least two body rows. Smaller tables are left as static — but still styled — tables.
- The filter field floats just above the table so revealing it never shifts the table (which keeps the sortable headers from moving out from under the cursor). If you wrap tables in a container with `overflow: hidden`, leave it room above — or use `overflow: visible` — so the field isn't clipped.

Layout modes

Every page ships with a left navigation sidebar and a right-hand “On this page” table of contents (TOC). When a page needs something different — more width, less chrome, or an edge-to-edge canvas — set `mode` in its front matter.

> **This page uses the default layout.** Nav on the left, TOC on the right, content in a comfortable reading column. The pages linked below each switch to a different `mode` so you can see the difference immediately.

The modes

`mode`	Left nav	Right TOC	Content width
(unset) / `default`	✓	✓	standard reading column
`wide`	✓	—	wider — good for big tables
`full`	—	—	wider, centered
`toc-only`	—	✓	wider — the hidden nav frees the room
`uncapped`	—	—	full-bleed: edge to edge, no width cap or padding

It’s a single front-matter line:

```
---
title: Release dashboard
mode: wide
---
```

See each mode live

- **Wide** — keeps the nav, drops the TOC, widens the column for big tables.
- **Full** — drops both sidebars; centered, distraction-free reading.
- **TOC only** — drops the nav, keeps the TOC for in-page jumps.
- **Uncapped** — full-bleed graphics, corner to corner.

Good to know

- Unrecognized values fall back to the default layout — a typo never breaks a page.
- Modes shape the **desktop** layout and compose with the responsive breakpoints: below 1100px the TOC hides and below 760px the nav hides, in every mode.
- The site header stays put in all modes, so there’s always a way back.

Wide mode

↑ All layout modes

> **This page uses `mode: wide`.** The left nav is still here, but the right-hand TOC is gone and the content column is wider than the default — so the table below has room for all its columns without wrapping or scrolling.

`wide` keeps the navigation handy while giving content more horizontal space. Reach for it on pages built around something wide: comparison matrices, data tables, wide diagrams, or side-by-side code.

Static site generators, compared

Tool	Language	Templating	Interactive UI	Data files	i18n	Search	API reference	Output
Aardvark	Python	Real Python in tags	Mantine React islands	JSON / YAML / CSV	Built-in	Built-in	Built-in (OpenAPI)	Static HTML
Docusaurus	JavaScript	MDX	Full React app	Plugin	Plugin	Algolia / local	Plugin	Static HTML + SPA
MkDocs Material	Python	Jinja2	Limited	None	Plugin	Built-in (lunr)	Plugin	Static HTML
Hugo	Go	Go templates	None	TOML / YAML / JSON	Built-in	Plugin	Plugin	Static HTML
Sphinx	Python	reST / Jinja2	Limited	None	Built-in	Built-in	autodoc	Static HTML + PDF
Eleventy	JavaScript	Nunjucks / Liquid / …	BYO	JSON / JS	BYO	BYO	BYO	Static HTML
Astro	JavaScript	Astro / JSX	Islands (any framework)	Content collections	i18n routing	Integration	Integration	Static HTML + SSR

That table has nine columns. In the default reading column it would wrap awkwardly or force a horizontal scrollbar; in `wide` mode it lays out comfortably.

How it composes

`wide` only changes the **desktop** layout. Narrow your browser below 1100px and the TOC is already hidden site-wide, so the page looks like any other — the nav collapses under 760px as usual.

Full mode

↑ All layout modes

> **This page uses `mode: full`.** Both the left nav and the right TOC are gone. What's left is a single centered column — nothing in the margins competing for attention. The site header stays, so you can always navigate away.

Some pages read best with nothing around them: a launch announcement, a release note, a long-form guide, an architecture write-up. `full` strips the chrome and centers the content at a comfortable measure so the words carry the page.

Why drop the chrome

A persistent sidebar is great for browsing a doc set, but on a page someone has arrived at to read end-to-end, it's visual noise. Removing it — and the TOC — narrows the focus to one thing. The content still uses a capped width (wider than the default reading column, but not edge-to-edge), because long lines of prose are tiring to read.

When to reach for it

- Announcements and changelogs that are read top to bottom.
- Marketing or overview pages that want a clean canvas but still need readable text.
- Any page where the global nav is a distraction rather than a help.

When not to

If readers need to jump between many sections of the same page, prefer `toc-only` — it drops the nav but keeps the "On this page" list. And if you want true edge-to-edge graphics, use `uncapped`.

Still reachable

Notice the header bar above is untouched: the logo, the tabs, search, and the light/dark toggle all stay. `full` removes the page's sidebars, not the site's navigation entirely — so a reader is never stranded.

TOC-only mode

↑ All layout modes

> **This page uses `mode: toc-only`.** The left nav is hidden so this single document owns the screen — but the right-hand “On this page” list stays. With the nav gone, the content column widens to fill the freed space. Scroll down and watch the TOC highlight the section you’re in; click any entry to jump.

`toc-only` fits a long, standalone reference — a configuration spec, an FAQ, a glossary, a policy — where the global nav is a distraction but moving within the page is essential. The headings below give the TOC plenty to work with.

Installation

Install the package, point it at a content directory, and run a build. The output is a folder of static HTML you can host anywhere.

Configuration

Every project has one config file at its root. It defines the site metadata, the navigation tree, the theme, and any integrations.

Site metadata

Name, description, and summary feed the page title, meta tags, and the generated `llms.txt`.

Navigation

The nav is a list of groups, each with a label and a list of links. It renders as the left sidebar — except, of course, on a page like this one.

Theme

Colors, fonts, and logos. Colors seed both the CSS variables and the Mantine island theme, so the chrome and the components stay in sync.

Content authoring

Pages are Markdown with optional front matter. Logic, when you need it, is real Python inside template tags.

Front matter

Title, description, keywords — and `mode`, which is how this page hid its nav.

Data files

Drop JSON, YAML, or CSV into the data directory and reference it by name from any page.

Components

Embed any Mantine component as an interactive island. Author once in Markdown; it mounts as React in the browser.

Internationalization

Author a base language at the root and translations under their own directories. A language picker appears automatically.

Search

Built in — a ⌘K search box scores a full-text index generated at build time, with no external service to configure.

Deployment

Build in CI and upload the output folder to any static host. Set the base URL so the sitemap and `llms.txt` use absolute links.

Wrapping up

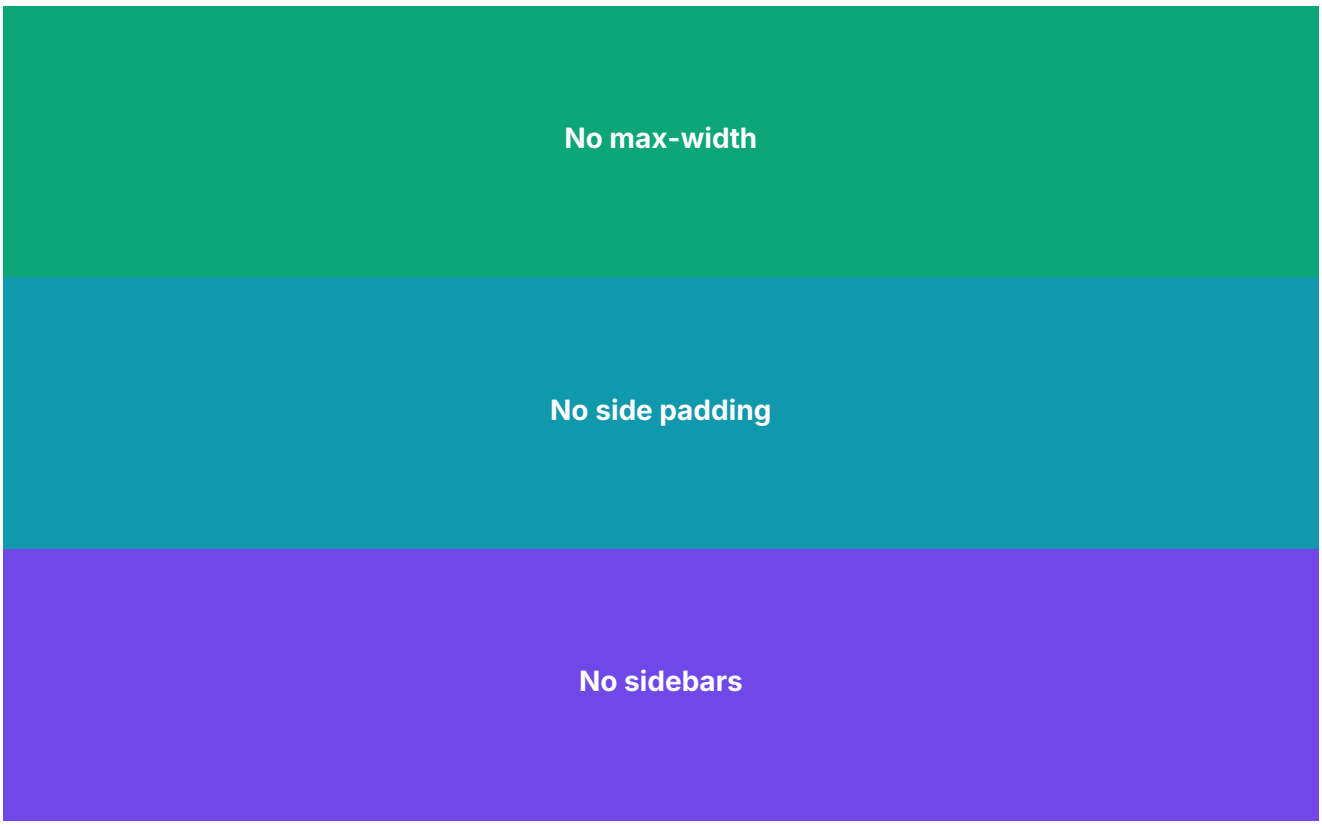
That’s nine headings and several subsections — enough that jumping around with the TOC genuinely beats scrolling. On a screen narrower than 1100px the TOC hides site-wide, so this page falls back to a plain single column.

Uncapped mode

mode: uncapped

Edge to edge

This hero reaches both edges of the viewport — no content width cap, no page padding. Ideal for landing pages and full-width graphics.



What you’re looking at

↑ All layout modes

This page sets `mode: uncapped`. Like `full` it hides both the nav and the TOC — but it goes further: it also removes the content’s max-width **and** the page’s padding. That’s why the hero and the colored band above run corner to corner, with no gutter on either side.

Because there’s no padding, **you** own the spacing inside an uncapped page. The bands above set their own `padding`; this explanatory section is wrapped in a centered container with its own `max-width` and padding so the prose stays readable. Mix full-bleed sections with contained ones however your design needs.

When to use it

- Landing and marketing pages with full-width hero imagery or video.
- Dashboards or visualizations that should fill every available pixel.
- Any page where the content is the graphic, edge to edge.

For a chromeless page that still keeps a readable, centered text column, reach for `full` instead.

Theme & customization

A **theme** is a self-contained folder of HTML templates plus the CSS, JS, fonts, and logos they use. Aardvark ships one theme, **`vark`** (the default docs theme), as **full, editable source** — there's no hidden layout. `vark new` copies it into your project's `themes/vark/` so you own it.

Where the theme lives

The bundled `vark` theme is packaged with Aardvark at `aardvark/themes/vark/`. `vark new` copies it into your project at **`themes/vark/`** — that's your editable copy, and a project-local `themes/<name>/` wins over the bundled theme of the same name, so you own the full source. Edit the files right there in `themes/vark/`.

For a one-off tweak where you don't want to fork the whole theme, drop a single file in **`templates/`** instead: any file there overrides the same-named file in the active theme (delete it to fall back). So `themes/vark/` is "I own this theme" and `templates/` is "I'm patching one file of it".

Theme CSS/JS/asset files must sit at the **top level** of the theme (or `templates/`) — only top-level files are emitted to `/_aardvark/`, with per-build fingerprinted filenames. Keep nested assets (images, icon sets, extra fonts) in `static/` instead, where the whole tree is copied and fingerprinted.

Files

- **`themes/vark/default.html`** — the page layout, rendered by the same `{% %}` engine as your content. It places the header, sidebar nav, content slot, TOC, and the "Was this page helpful?" widget. **Every theme must provide a `default.html`** — it's the layout used for any page that doesn't pick another.
- **`themes/vark/theme.scss`** — the color system and chrome styles. The `$`-variables at the top are the single source of truth for every color (light + dark); compiled to `/_aardvark/theme-<sha>.css` at build time. See Colors. In source templates, keep writing `/_aardvark/theme.css`; Aardvark rewrites it during the build.
- **`themes/vark/color-scheme.js`** — light/dark handling (see Light & dark mode).

Selecting a theme

The active theme is `theme.name` in `aardvark.config.yaml`; it defaults to `vark`:

```yaml
theme:
  name: vark            # the default — a bundled theme, or one you ship yourself
```

To ship your own theme, drop its folder at `themes/<name>/` in your project (it must contain a `default.html`) and point `theme.name` at it. A project-local `themes/<name>/` wins over a bundled theme of the same name; an unknown name warns and falls back to `vark`, so a typo never breaks

the build. Per-file `templates/` overrides still apply on top of whichever theme is active.

Page templates

Every page uses the theme's `default.html` unless it asks for another. A theme can hold any number of templates beside `default.html` (e.g. `landing.html`, `blog.html`); a page opts into one with a `pagetype:` front-matter key:

```
---
title: Launch
pagetype: landing      # renders with the theme's landing.html
---
```

`pagetype: landing` resolves `landing.html` through the same cascade as the default layout (your `templates/landing.html` first, then the active theme's). A `pagetype:` that names a template the theme doesn't have warns at build time and falls back to `default.html`, so a misspelling degrades gracefully instead of failing.

This site ships a `landing.html` in its theme and a page that opts into it. Here's the whole template, read straight from the theme file so it never drifts from what actually renders:

themes/vark/landing.html

```
<!DOCTYPE html>
<!--
  A sample alternate theme template, selected per page with `pagetype: landing` in
  front matter. It demonstrates that a theme can carry more than one layout: this one
  drops the sidebar nav and the on-this-page TOC and centers the content in a full-width
  hero, while reusing the same `default.html` context (brand, theme styles, assets).
  Lives only in this project's themes/vark/ (not the packaged vark theme), so it's a
  worked example of "add your own template and point a page at it". The hero styling is
  scoped here and built on the theme's --aardvark-* variables, so it stays on-theme in
  both light and dark mode without touching theme.css.
-->
<html lang="{% lang %}"{% (' data-aardvark-banner="' + banner_id + '"') if banner_id else ''
%}>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>{% full_title %}</title>
  {% seo_head %}
  <script src="{% asset('/_aardvark/color-scheme.js') %}"></script>
  {% ('<script src="' + asset('/_aardvark/banner.js') + '"></script>') if banner_id else ''
%}
  <link rel="stylesheet" href="{% asset('/_aardvark/theme.css') %}" />
  {% theme_style %}
  {%
  for href in css_files:
      page.print('<link rel="stylesheet" href="' + href + '">')
  %}
  <style>
```

```
    .aardvark-landing-main { display: flex; justify-content: center; padding: 4rem 1.5rem; }
    .aardvark-landing.aardvark-has-site-footer .aardvark-landing-main { padding-bottom:
calc(4rem + var(--aardvark-footer-gap)); }
    .aardvark-landing-hero { max-width: 52rem; width: 100%; text-align: center; }
    .aardvark-landing-hero h1 { font-size: clamp(2rem, 5vw, 3rem); margin-top: 0; }
  </style>
  {% head_extra %}
</head>
<body class="aardvark-landing{% ' aardvark-has-site-footer' if footer_html else '' %}">
  <a class="aardvark-skip-link" href="#aardvark-main">{% t("Skip to main content") %}</a>
  {% banner_html %}
  <header class="aardvark-header">
    {% brand_html %}
    <button type="button" class="aardvark-theme-toggle" aria-label="Toggle color scheme">
      <span class="aardvark-icon-light">&#9788;</span><span
class="aardvark-icon-dark">&#9789;</span>
    </button>
    {% top_buttons_html %}
  </header>
  <main id="aardvark-main" class="aardvark-landing-main" tabindex="-1">
    <article class="aardvark-content aardvark-landing-hero">{% content %}</article>
  </main>
  {% footer_html %}
  {% powered_by_html %}
  <div id="aardvark-live" class="aardvark-visually-hidden" role="status"
aria-live="polite"></div>
  <script src="{% asset('/_aardvark/code-blocks.js') %}" defer></script>
  {%
  for src in js_files:
      page.print('<script src="' + src + '" defer></script>')
  %}
  {% body_extra %}
</body>
</html>
```

See it live

The landing example renders with `landing.html` instead of `default.html` — same theme, a different layout (no sidebar, a centered hero).

What the layout receives

`default.html` (and any `pagetype:` template) is rendered with these variables (in addition to the usual `data`/`site`/`config`/`page`):

Variable	Contents
`content`	The rendered page HTML

`not_found_html`	The 404 page's "Were you looking for…" search suggestions island; `""` on every other page (see below)
`title`	The page title
`nav_html`	The page's isolated left nav, built from front-matter menus
`breadcrumbs_html`	The breadcrumb trail (a Mantine island), or `""`
`toc_html`	The on-this-page table of contents
`page_url`	The current page's URL
`css_files` / `js_files`	Your root CSS/JS assets, already rewritten to fingerprinted build URLs
`head_extra` / `body_extra`	Integration + islands tags

Because it's the same engine, you can use Python, `component(...)`, and `asset(path)` in the layout too — e.g. embed a Mantine component in the header or construct a dynamic URL to a fingerprinted static asset.

The built-in `vark` theme renders `not_found_html` just before `content` (`{% not_found_html %}{% content %}`). It's empty on normal pages, so it's harmless everywhere — but a **fully custom `default.html` must include the `{% not_found_html %}` slot** to get the search-powered 404 suggestions (see Search → 404 suggestions). Omitting it degrades gracefully: the 404 page still shows your static `404.md`, just without the "Were you looking for…" island.

Nav

The sidebar comes from the `nav:` tree in `aardvark.config.yaml`: groups with `items`, each item a `label` + `link`. The current page is highlighted automatically.

Breadcrumbs

Every page (except the home page) gets a breadcrumb trail at the top of the content, rendered with Mantine's Breadcrumbs component. The trail is **menu-based**: Aardvark finds the current page in your `nav:` tree and shows the path of menu labels to it — e.g. a page under the Buttons & actions group shows `Home / Buttons & actions / Button`. Pages absent from the nav fall back to `Home / <page title>`.

It's on by default. Turn it off — or tune it — with a top-level `breadcrumbs:` block in `aardvark.config.yaml`:

```
breadcrumbs: false            # disable entirely
```

```yaml
breadcrumbs:
  enabled: true               # default
  home: true                  # prepend a linked Home crumb (default true)
  homeLabel: Home             # its label (localized like nav labels)
  separator: "/"              # Mantine's separator (defaults to "/")
```

Individual pages can opt out without touching the config: set `breadcrumb: false` in a page's front matter to hide the trail on just that page. The Changelog does this, since a `Home / Changelog` crumb adds nothing above its title. This hides only the visible trail — the page's hover-card preview and its `BreadcrumbList` structured data still reflect its place in the hierarchy.

Labels and links are localized per language just like the sidebar nav. The trail is rendered into the `breadcrumbs_html` layout variable, so editing `themes/vark/default.html` lets you move or restyle it.

Colors

Every color on the site is declared once, in your theme's **`themes/vark/theme.scss`** — the single source of truth. Run `vark new` to get an editable copy of the theme in your project, then edit the `$`-variables at the top of `theme.scss`. The file is compiled to `/_aardvark/theme-<sha>.css` at build time (with Sass), and the rest of the stylesheet only ever references those variables — so changing one value recolors the whole site. Your theme source can still link `/_aardvark/theme.css`; the output HTML is rewritten to the fingerprinted URL.

Each color is a **light/dark pair**, so both schemes are always covered:

```scss
// Chrome
$bg-light:      #ffffff;   $bg-dark:      #1a1b1e;   // page background
$fg-light:      #1a1b1e;   $fg-dark:      #c9c9c9;   // body text
$surface-light: #f5f3ff;   $surface-dark: #221d33;   // cards & panels

// Brand — primary also drives links, focus rings & active nav, and seeds the Mantine island
// palette, so color="primary" / color="secondary" resolve in components.
$primary-light:   #7048e8;  $primary-dark:   #a78bfa;
$secondary-light: #0ca678;  $secondary-dark: #38d9a9;

// Code & tables get their own knobs. The default theme keeps them a neutral grey ($neutral,
// below) rather than the brand-tinted $surface used for cards — retint freely.
$neutral-light: #f8f9fa;           $neutral-dark: #25262b;
$code-bg-light:         $neutral-light;  $code-bg-dark:         $neutral-dark;  // fenced
code blocks
$code-inline-bg-light:  $neutral-light;  $code-inline-bg-dark:  $neutral-dark;  // `inline
code`
$table-stripe-light:    $neutral-light;  $table-stripe-dark:    $neutral-dark;  // zebra
rows
$table-header-bg-light: $fg-light;       $table-header-bg-dark: $fg-dark;        // header
band
```

Because it's real Sass, you can **derive** shades instead of hand-picking each one — the accent fill is just `rgba($primary-light, 0.12)`, so it tracks the brand automatically. The full set of variables —

chrome, brand, code, tables, diff, plus advanced semantic colors (API method chips, status dots, shadows) — sits grouped and commented at the top of `theme.scss`.

The brand `primary` link color is checked by the build-time contrast audit, so custom-brand links stay WCAG-readable on both schemes.

> **Moved from config.** Colors used to be set under `theme.colors` in `aardvark.config.yaml`. That's been replaced by `theme.scss`; a `theme.colors` block in config is now ignored (the build warns and points you here).

Page background

Give the whole page a background image with `theme.backgroundImage`. Drop the file in `static/` and point at it — the image layers over the background color and shows through behind the content, sidebar, and TOC (the header keeps its solid band):

```
theme:
  backgroundImage: /backgrounds/page.svg   # one image, sensible defaults
```

Use a mapping for a separate dark-mode image and to control placement:

```
theme:
  backgroundImage:
    light: /backgrounds/light.svg   # light mode (also used in dark unless `dark` is set)
    dark: /backgrounds/dark.svg     # optional dark-mode image
    size: cover                     # how it's scaled
    position: center                # where it's anchored
    repeat: no-repeat               # whether it tiles
    attachment: scroll              # scroll with the page, or `fixed` to stay put (desktop
only)
```

Each knob takes a friendly word **or** any raw CSS value, so you can do whatever CSS allows:

Knob	Options	What it does
`size`	`cover` / `fill` / `scaled` · `fit` (contain) · `stretch` · `original` · a length like `400px`	How the image is scaled. `cover` fills the page, cropping overflow; `fit` shows all of it.
`position`	`center` · `top` · `bottom` · `left` · `right` · pairs like `right top` · `50% 20%`	Where the image is anchored.
`repeat`	`no-repeat` / `once` · `tile` / `tiled` (repeat) · `horizontal` · `vertical`	Whether and how it tiles.

`attachment`	`scroll` (default) · `fixed` · `local`	`scroll` moves the image with the page; `fixed` keeps it still (a parallax look) — but `fixed` is **unsupported on iOS Safari**, so it's a desktop-only opt-in.

The defaults (when you set only an image) are `cover` · `center` · `no-repeat` · `scroll` — a full-bleed image. Add `attachment: fixed` for a parallax background that stays put as you scroll, but only target desktop with it: iOS Safari doesn't support fixed backgrounds. For a small repeating texture, set `size: original` and `repeat: tile` instead.

The image also extends up through the sticky header and tab bar — but only with `attachment: fixed`, where it's viewport-locked and lines up seamlessly with the page. Under the default `scroll` the bars stay solid (the image would otherwise be sized to each bar's own box and seam at the edge), and the background lives on the page body alone.

Favicon

Set the browser-tab (and bookmark) icon with `theme.favicon`. Drop the file in `static/` and point at it — the extension sets the `<link type>`, so an SVG scales to any tab size:

```yaml
theme:
  favicon: /favicon.svg
```

Ship several formats by passing a list — the browser picks the best one it can render, which is handy for a legacy `.ico` fallback alongside a scalable SVG:

```yaml
theme:
  favicon:
    - /favicon.svg
    - /favicon.ico
```

Leave it unset and the browser just probes `/favicon.ico` on its own.

Powered by Aardvark

Every page ends with a small **Powered by Aardvark** footer linking to aardvarkdocs.com. Its logo swaps with the active light/dark scheme, just like the header brand.

It's on by default. Sites with build-time AI enabled — any `ai:` feature — can remove it from `aardvark.config.yaml`:

```yaml
poweredBy: false              # honored on the AI/paid tier
```

On the free tier the link is kept — the attribution is genuinely appreciated — so `poweredBy: false` there is ignored, and the build prints a friendly note. Either way the footer is wrapped in a stable id, so you can always hide it with one line of CSS (in a root stylesheet or `themes/vark/theme.scss`):

```css
#aardvark-powered-by { display: none; }
```

Syntax highlighting

Fenced code blocks are highlighted at build time with Pygments — no client-side JavaScript. Choose a color theme for each scheme under `theme.syntax`:

```
theme:
  syntax:
    theme: one-light       # light-mode preset
    themeDark: one-dark    # dark-mode preset
```

Set `enabled: false` under `theme.syntax` to turn highlighting off entirely — code blocks then render as plain `<pre><code>` with no token coloring.

`theme` and `themeDark` accept any Pygments style — for example `monokai`, `dracula`, `nord`, `github-dark`, `solarized-light`, `solarized-dark`, or `gruvbox-dark` — plus the built-in `one-light` / `one-dark` pair used by default. A dark preset also sets the code block's background so themes like Monokai look right.

To fine-tune, override individual token colors by name (these win over the preset). Light colors go under `colors:`, dark under `colorsDark:`:

```
theme:
  syntax:
    colors:                # light
      keyword: "#a626a4"
      string: "#50a14f"
      comment: "#a0a1a7"
    colorsDark:            # dark
      keyword: "#c678dd"
      string: "#98c379"
```

The recognized token names are `comment`, `keyword`, `constant`, `string`, `number`, `function`, `builtin`, `decorator`, `class`, `variable`, `attribute`, `tag`, `operator`, and `heading` (Markdown headings inside a code sample).

Code block labels

Tag each fenced block with its language so it highlights correctly — `python`, `yaml`, `bash`, `json`, and so on. Aardvark template examples have a dedicated `aardvark` label that colors the `{% %}` directives — tag and attribute names and their values — and renders the surrounding Markdown body.

Shell fences (`bash`, `sh`, `shell`, `zsh`) color the command name, flags, variables, and `NAME=value` assignments — more than Pygments' stock bash lexer, which leaves bare commands and their arguments uncolored.

Light & dark mode

The default theme includes a light/dark toggle (the sun/moon button in the header). Try it — this page updates instantly.

How it works

Mantine v7 styling is **attribute-driven**: components read their colors from CSS variables keyed on `data-mantine-color-scheme` on `<html>`. So switching themes is just flipping that attribute — no React re-render needed.

- **`themes/vark/color-scheme.js`** runs in `<head>` before paint. It reads the saved preference from `localStorage['aardvark-color-scheme']` (or the OS setting when unset) and sets the attribute, avoiding a flash. It also wires the `.aardvark-theme-toggle` button.
- **`themes/vark/theme.scss`** maps the chrome's colors onto Mantine's own variables (`--mantine-color-text`, `--mantine-color-default-border`, …), so the chrome and the islands always match — in both schemes.

Because the chrome reuses Mantine's variables, you rarely need scheme-specific CSS. If you do, target the attribute:

```
:root[data-mantine-color-scheme="dark"] .my-thing { /* dark-only */ }
```

One caveat when the dark rule changes a **background**: the default theme cross-fades page navigations with a view transition, and the browser draws the incoming page's first frame before `color-scheme.js` has set the attribute — so an attribute-keyed rule doesn't apply to that frame, and dark-mode readers see your light background flash for the length of the fade. Give such rules an OS-scheme fallback too. Because those bare dark rules keep matching after the attribute is set, add an explicit-light override alongside them, so a reader who chose light mode on a dark OS gets their choice back the moment the attribute lands (that reader's pre-attribute frame is never shown — the theme skips the cross-fade whenever the chosen scheme differs from the OS):

```
@media screen and (prefers-color-scheme: dark) {
  .my-thing { /* dark, for the pre-attribute first frame */ }
  :root[data-mantine-color-scheme="light"] .my-thing { /* light wins back */ }
}
```

(Keep the `screen` type if your site builds the whole-site PDF — the PDF renderer applies bare feature queries but skips `screen` blocks, which keeps the PDF on the light palette. This is the pattern for **page-local CSS**; the theme's own chrome closes the same first-frame gap with different machinery — an attribute-less fallback block in `theme.scss` plus the view-transition skips in `color-scheme.js`. This site's homepage source is a worked example of the page-local pattern.)

Default

With no saved preference the site follows the operating system, and updates live if the OS theme changes. Once a visitor clicks the toggle, their choice is remembered.

A different template, same theme

This page sets `pagetype: landing` in its front matter, so Aardvark renders it with `themes/vark/landing.html` instead of the usual `default.html`. Notice there's no sidebar and no on-this-page rail — the landing layout drops them and centers the content.

Everything else is the same theme: the header, brand, colors, and footer all come from the `vark` theme's shared chrome. A template is just another layout in the theme; a page picks one with `pagetype:`, and anything without it gets `default.html`.

See Theme & customization for how to add your own templates.

Search

Aardvark has **built-in search** — no Algolia, no crawler, no API keys. The build writes a full-text index of your site and a Mantine search box scores it in the browser as you type.

Using it

Press ⌘K (or `/`) anywhere, or click the search box in the header. Then:

- type to see ranked results update live, with the matching text highlighted so you can tell why each result surfaced;
- after results appear, use the path filter to narrow the list to one or more sections of the site;
- leave **Exact matches** off for small spelling slips, or turn it on when you want only exact typed terms;
- use ↑/↓ to move through results and ↵ to open the highlighted one;
- press `Esc` to close.

Narrowing a search

Two operators help when a plain keyword search returns too much:

- **Exact phrase** — wrap words in double quotes to match them as one literal, consecutive phrase: `"getting started"` finds only pages with that exact wording (punctuation counts), not pages that merely mention getting and started apart.
- **Exclude a word** — prefix a term with a minus sign to drop every page that contains it: `theme -dark` finds theme pages that don't mention dark. You can exclude a phrase too — `-"under construction"`. Exclusions look at page content, not the URL or breadcrumb, so `-dark` won't hide a page that just happens to live under `/dark-mode/`.

Mix them freely: `"api key" -deprecated` requires the exact phrase api key and drops anything mentioning deprecated.

The search box also corrects small typos for unquoted terms by looking at high-signal labels: titles, headings, and keywords. That helps a misspelling still reach the right page, but it can be too generous for short product/component names. Check **Exact matches** in the search dialog to require typed terms exactly. Quoted phrases and exclusions are always literal.

Filtering by path

After a query has results, the search dialog also shows a **path filter**. It is built automatically from the URLs in the current result set: if the results include `/components/inputs/textinput/`, the filter can offer `/components/` and `/components/inputs/`. Root (`/`) is skipped so the menu stays focused on useful site sections.

Pick one or more folders to keep only results under those paths. For example, `/components/inputs/` keeps input component pages, while `/components/` keeps the whole components section. Multiple selected paths are combined as an OR filter: choosing `/components/buttons/` and `/authoring/` shows results from either area.

Path filters do **not** rerank results. The search scorer still runs once for the query, then the selected paths narrow the visible list while preserving that ranking order. Matching is path-boundary aware, so `/components/` matches `/components/button/` but not `/components-extra/`.

Filters stay selected while you refine the query, as long as the new result set still contains those paths. If a selected path disappears, Aardvark drops that selection so a stale filter cannot make a valid search look empty. Closing the search dialog clears both the query and selected paths.

What the build does

Every build writes `search-index.json` to your site root — one record per page:

field	what it holds
`url`, `title`, `breadcrumb`	where the page lives and what it's called
`description`, `keywords`	from the page's front matter
`headings`	every `h1`–`h6` on the page
`inboundLinks`	a `{text: count}` map of the link text other pages use to link here — deduped, with how many times each is used
`text`	the full, user-visible body text (markup stripped)

The client scores matches by where they hit: a hit in the **title**, **URL**, or **breadcrumb** is worth 10; in the **headings**, **keywords**, or **description**, 5; and anywhere in the **body**, 1. A match in the **inbound link text** is worth 5 too, scaled up (dampened, by `log2` of the count) when many pages link here with that text — so a widely-referenced page ranks a little higher for the words others use to point at it, without letting nav links dominate. A multi-word query that appears **verbatim** in the title, a heading, or the description earns an extra bonus on top, so exact phrase matches in those summary fields rank above incidental word hits — and a `"quoted phrase"` (see above) is required, matched literally anywhere on the page, not merely boosted. Results sort by total score, strongest first; when two pages score the same, the one with more headings wins (a fully documented page beats a one-line stub), then the more-linked page, then title.

Matching is **accent-insensitive**: the query and the index are folded the same way (diacritics stripped, case ignored), so searching `cafe` finds Café and `münchen` finds Munchen — either spelling matches either. This is always on and needs no config.

Every one of these weights is configurable — see Tuning the ranking.

Per-page control

Two front-matter keys tune how an individual page behaves in search:

- **`searchable: false`** removes the page from the search box and every AI/agent corpus the build emits — the assistant / MCP index, `llms.txt`, and page cards — while leaving it **fully public and crawlable**: it stays in the sitemap, keeps its normal `robots` meta, and still serves, so search engines index it as usual. Use it to declutter results with pages readers shouldn't stumble into via search (a legal boilerplate page, a redirect stub). This is distinct from `noindex`, which goes further and hides the page from the outside world too — dropping it from the sitemap and emitting a `noindex` robots directive so search engines skip it.
- **`boost: <number>`** scales a page's match score (default `1`). It's multiplicative, so it reranks pages the query already matches rather than forcing an unrelated page to the top — `boost: 2` roughly doubles a page's rank strength, `boost: 0.5` halves it, and `boost: 0` soft-hides a page (it stays indexed but can never rank). Handy to float a canonical "Getting started" above near-duplicates, or sink a deprecated page.

```
---
title: Deprecated API
boost: 0.2          # still findable, but ranks below the current docs
---
```

Tuning the ranking

Set any of the weights under `search.ranking` in your config. Anything you leave out keeps its default, so you only list what you want to change. These are the defaults:

```
search:
  ranking:
    title: 10        # match in the page title
    url: 10          # match in the URL path
    breadcrumb: 10   # match in the breadcrumb trail
    headings: 5      # match in any h1-h6 heading
    keywords: 5      # match in the front-matter keywords
    description: 5   # match in the front-matter description
    text: 1          # match anywhere in the body text
    inboundLink: 5   # match in other pages' link text (then scaled up by how many link
here)
    phrase: 10       # bonus for a verbatim multi-word phrase in the title, a heading, or
the description
    synonym: 0.5     # a synonym match scores 50% (0.5×) of the same word matched exactly;
see Synonyms
    typo: 0.5        # a typo-corrected match scores 50% (0.5×) of an exact match; see Typo
tolerance
```

The last two — `synonym` and `typo` — are **multipliers**, not field weights. A synonym- or typo-matched term earns its normal per-field score, then that score is **multiplied by this factor**. So

the default `0.5` means such a match is worth **half** of the same word matched exactly: if an exact title match is worth `10`, the same word reached through a synonym or a typo-correction is worth `5`. That's why synonyms and typo-corrected matches rank below an exact match.

Set it to `0.25` to count them at a quarter, `1` to count them the **same** as an exact match (no penalty), or `0` to keep the expansion for recall but give it no ranking weight at all. Keeping the factor **below** `1` preserves the rule that an exact match always wins; a value **above** `1` would let a synonym or fuzzy match outrank a literal one.

Set a weight to `0` to switch a field off entirely. You only list the weights you want to change — everything else keeps its default. A few common adjustments:

Rank on titles and structure, not prose. Good when bodies are long and repetitive (API references, generated docs) and the title already says what a page is:

```yaml
search:
  ranking:
    text: 0       # ignore body-text matches
    headings: 8   # lean harder on section headings
```

Reward exact wording. Boosts pages where the query appears verbatim in a title, heading, or description — handy for a glossary or a precise term index:

```yaml
search:
  ranking:
    phrase: 25    # a verbatim multi-word match is a strong signal
```

Trust your own titles over how other pages link. Lower the inbound-link weight when a few heavily-linked hub pages crowd out more relevant results:

```yaml
search:
  ranking:
    inboundLink: 1
```

Lean on synonyms and typo-correction. If your synonym map is curated and your terms are easy to misspell, let those matches compete closer to exact ones (still below, so an exact match wins):

```yaml
search:
  ranking:
    synonym: 0.9   # a synonym match counts almost as much as an exact one
    typo: 0.7      # forgive misspellings more generously
```

Validation. Weights and factors must be numbers `>= 0`. An unknown key, a non-numeric value, or a negative one is ignored with a build warning and falls back to its default — so a typo can't silently skew ranking. The tie-breaking order used when two pages score equally (more headings, then more inbound links, then title) isn't configurable.

Typo tolerance

Search forgives misspellings out of the box: a query term that matches nothing is compared against the words in your titles, headings, and keywords, and the closest ones (within one edit for short words, two for longer) are searched too — so `instalation` still finds Installation. Corrected matches rank **below** exact ones, so a real match is never buried — and you control how far below with the `search.ranking.typo` factor (see Tuning the ranking). It's **on by default**; tune or disable it under `search.typoTolerance`:

```
search:
  typoTolerance: false        # turn it off entirely
```

...or tune it:

```
search:
  typoTolerance:
    maxDistance: 1            # max edits to consider (0–2; default 2)
    minLength: 5              # don't fuzz terms shorter than this (default 4)
```

Short terms and API symbols are left alone (`minLength`) so they aren't fuzzed into unrelated words.

Synonyms

Map the words readers type to the words your docs use — so a search for one finds pages that only mention the other. Each entry is **multi-way**: every word in a group matches every other.

```
search:
  synonyms:
    otp: [one-time password, 2fa code]
    login: [sign in, signin, log in]
    k8s: [kubernetes]
```

With this, `otp` also finds pages that say one-time password — and searching one-time password finds pages that only say `otp`. Matching is by whole query word: a single-word synonym (like `k8s`/`kubernetes`) works in **both** directions, and a multi-word value matches when the reader types its words in order, so it too expands back to the rest of its group. Synonyms expand the reader's query only — they rank below an exact match (tune how far below with the `search.ranking.synonym` factor — see Tuning the ranking), and a `"quoted phrase"` or `-exclusion` is never expanded (those are explicit literal signals). Checking **Exact matches** also skips synonym expansion for that search. Malformed entries are dropped with a build warning.

Section & API-symbol results

Turn this on to let a result jump straight to the **heading** it matched, instead of the top of the page — and to make **API operations searchable by symbol**:

```
search:
  sections: true          # → default heading band h2–h3
```

...or choose the band:

```yaml
search:
  sections:
    minLevel: 2
    maxLevel: 4
```

With `search.sections` on:

- a match in a section heading deep-links to that heading (`/guide/auth/#tokens`), with the heading shown in the result's trail;
- every `{% openapi %}` operation becomes findable by its **operationId**, its `METHOD /path`, or its tag — so a search for `createPet` or `POST /pets` lands right on that operation. (Operations are rendered by the API-reference island, so they're invisible to the normal page-text indexer without this.)

It's **off by default** because section records grow the index; a match in body prose still surfaces the page even when it's off. This composes with the path filter — faceting still narrows by site area, while sections locate the spot within a page.

Multilingual sites

On a site with more than one language, each result is tagged with its page's language, and the search box gently **prefers results in the language you're currently reading** — a translated page ranks above its foreign-language twin, while an untranslated page stays findable. No configuration required. Matching is also accent-insensitive (see above), which helps across languages that share a script.

404 suggestions

When search is on, the generated **404 page** helps a lost reader recover: it reads the URL that missed, searches the index for the closest pages, and lists them as "were you looking for…" links (deep-linked to a heading when sections are on). Because it reuses the search scorer, typo tolerance quietly fixes a fat-fingered URL too — `/instalation/` suggests Installation. It's automatic; just write a helpful `404.md`, which stays the fallback when a visitor has JavaScript disabled. The header ⌘K search is available on the 404 page as well.

The built-in theme wires this in for you. If you ship a **fully custom theme**, add the `{% not_found_html %}` slot to your `default.html` (the built-in places it just before `{% content %}`) to get the suggestions island — see Theming → what the layout receives. Without it the 404 page still renders your static `404.md`; you just don't get the "were you looking for…" island.

Analytics

When the Google Analytics integration is configured (`integrations.analytics`), search activity is reported automatically — no extra setup:

- a **`search`** event (with `search_term` and the number of `results`) each time a query settles. This feeds GA4's built-in Site search report, and the `results: 0` events show what people look for but don't find — your content gaps.

- a `search_select` event (`search_term`, `url`, `position`) when a result is opened, so you can see which queries lead where.

With analytics off, nothing is sent.

Search Analytics dashboard (AI-enabled sites)

If your site uses the built-in AI assistant (a metered “Aardvark cloud” key), the search box also reports activity to your dashboard’s **Search Analytics** tab — a superset of what Fern, ReadMe, and Mintlify show:

- **top searches** and **zero-result queries** (your content gaps), with a volume trend;
- **click-through rate**, **average result position**, and **no-click sessions** (searches that went nowhere);
- **Ask-AI escalation rate** — how often a search turned into an assistant question;
- **top clicked results**, a **language** breakdown, and **search latency**;
- periodic deterministic **search themes** and **content gaps** summarized per normalized term from a bounded recent sample (up to 50,000 stored queries and the latest 50,000 click candidates considered per source period; clicks need not duplicate their query text and count only when their exact nearest owner is one of those sampled queries), with concrete documentation recommendations that remain available after the raw event-retention window;
- **CSV export** of the top terms.

Query text is stored by default (bounded; the base raw-row retention window is **90 days by default** and configurable via the gateway’s `SEARCH_EVENT_RETENTION_DAYS`). When that window is shorter than the active calendar period, the gateway retains only the latest 50,000 stored queries, latest 50,000 click candidates, and at most one preceding stored-query owner per retained click for each account — no more than 150,000 extended source rows per retained source period, and normally fewer. This is the same bounded source used to rebuild the active theme/gap snapshot. If activity arrives after a period’s last refresh, at most one recently completed pending period keeps its own bounded sample while the fair queue finalizes that tail; later expired backlog periods are dropped. Active plus pending samples therefore cap transient extended storage at 300,000 rows per account. Anonymous queries and Ask-AI rows still follow the base window. Lowering `SEARCH_EVENT_RETENTION_DAYS` makes older rows eligible immediately, but does not purge them synchronously: the oldest-first minute sweep drains bounded pages fairly across accounts, so a large existing backlog ages out gradually. After those source rows expire, the live term-level drill-down ages out while a previously generated theme/gap snapshot can remain. Those snapshots cover their latest calendar period independently of the dashboard’s selected 24-hour/7/30/90-day live window. The live KPIs and volume trend are computed from those raw rows, so they age out with the rows rather than becoming a separate permanent aggregate. The durable summary applies its source-row cap before grouping, so displayed theme/gap counts are counts within that recent sample when an account or period is busier than the cap. Result clicks follow the nearest preceding stored-term query in the same tab session, whether or not the click beacon repeats the term. A sampled click contributes only when that exact owner is also in the capped query sample; an omitted or out-of-period owner remains as an attribution barrier so

retention cannot lend the click to an older sampled query. A click affects a durable snapshot only when its raw beacon timestamp falls inside the owning query's calendar period plus the five-minute boundary/reordering window; a later same-tab click remains part of live analytics but does not reopen or extend that snapshot. Anonymous queries stay outside this durable term-insight attribution. If that source period's click-candidate lane reaches its cap and a sampled query has no attributed click, or a click candidate for the same normalized term belongs to a same-period owner outside the capped query sample, its click count is treated as unknown rather than manufacturing a no-click content gap. The same conservative unknown applies when a delayed query takes ownership of a click below its new period's five-minute window: that older click is not counted in the new period, but it cannot manufacture an exact no-click gap there. A durable snapshot retains a representative query-term label and example for each normalized term bucket, and a content-gap recommendation may quote that label, after the underlying raw rows are deleted. The active-period snapshot remains alongside the source-row floor described above. After that period closes, the scheduled sweep removes snapshots whose generation time falls outside `SEARCH_EVENT_RETENTION_DAYS`, bounding derived query text to the active period plus the configured window. Snapshots can survive `/admin/delete-account`, which archives the account, only until that cutoff; an underlying hard delete removes them immediately. Set `search.analytics.store_terms: false` to log only the anonymous funnel (counts, rates, positions, latency — no raw text), or `search.analytics: false` to turn the dashboard capture off entirely. Readers with **Do-Not-Track** or **Save-Data** set are never logged by the shipped search widget — it sends no beacon (a client-side check, like `store_terms` above). (This is independent of the GA events above — you can run either, both, or neither.)

Search-theme and content-gap summaries require stored query terms. With `search.analytics.store_terms: false`, the dashboard can show the anonymous live funnel while its raw events remain. After the rebuilt setting is deployed, the shipped widget omits term text from newly emitted events, so those events cannot seed a term-based theme or gap. The setting does not erase term-bearing events already received or snapshots already written: eligible earlier rows can still be summarized before they expire, and existing snapshots remain until their active-period/retention cutoff.

`store_terms: false` is a **client-side, best-effort** control: your site's search widget omits the query text before sending, so with the shipped widget a reader's query text isn't transmitted. The gateway does **not** re-check the flag — it stores whatever `term` a caller sends — so it isn't an absolute guarantee: a custom or scripted client that ignored the setting could still write term text. On the server, the ingest endpoint's **origin allowlist** is the primary safeguard: it only accepts beacons whose `Origin` is your own docs site, so the world-readable `aardvark_live_` key in `ai-config.json` can't be used from another site's browser to write into your analytics — that allowlist, not the per-account rate cap, is what keeps a leaked public key low-risk. It isn't an absolute gate (a determined caller scripting requests outside a browser can forge the `Origin` header), but even then they can only inject their own text (bounded and rate-capped), never a reader's query. In short: the widget + `store_terms` protect readers' text client-side, and the origin allowlist is the server-side backstop that keeps other sites out.

Search analytics answers "what did readers look for". Its companion, Content Reach (Business and Enterprise), answers "which parts of the page did they get to once they landed" — including how often a search result click leads to a section actually coming into view. The two use **separate**

tab-session ids on purpose, and no query text ever crosses into Content Reach. They do both record the page path and a timestamp, though, so on a quiet page the two datasets can still be lined up by someone reading both — see Content Reach's Privacy section.

Index size

The index is gzipped at build time — `search-index.json.gz` is written next to the plain file, and the search box gunzips it in the browser (via `DecompressionStream`), so it downloads small on **any** host, not only ones that apply `Content-Encoding` themselves (typically a 3–4× saving). Browsers without `DecompressionStream` fall back to the plain `search-index.json` automatically.

The JSON itself is minified (it's read by code, not people), so the plain file stays as small as possible before gzip even kicks in.

To skip the gzipped copy and rely on your server's own compression instead:

```
search:
  compress: false
```

Caching

The index is served from a stable path (`search-index.json` / `.gz`), so make sure your host serves it with **revalidation** — an `ETag`/`Last-Modified` (most static hosts and CDNs do this by default) or `Cache-Control: no-cache` — so readers pick up fresh results right after a redeploy instead of a stale cached copy. It only needs attention if your host sets long, non-revalidating cache lifetimes on JSON.

> **Serve from the domain root.** The search box fetches `/search-index.json` by an absolute, root-relative path — as the theme does for all its generated assets (the islands bundle, `theme-<sha>.css`, hover-card data, …). Deploying under a URL sub-path (e.g. `https://example.com/docs/`) isn't supported: those assets would 404. Host the site at the origin root (or a subdomain), which is the default for the supported targets (Cloudflare, Netlify, GitHub Pages project sites via a subdomain).

Turning it off

Search is on by default. To disable both the header box and the index file, set:

```
search: false
```

> The search box is a Mantine island, so it appears on builds that bundle islands (Node + esbuild — the default). `search-index.json` is written either way, so a `--no-bundle` build can still be searched once rebuilt with the bundle.

Content Reach

Page views tell you a page was opened. They don't tell you whether anyone got as far as the install command. **Content Reach** answers that: for every page, which of its sections came into view, how long each held the reader's position, and which controls readers actually used.

It reports on your page's own **sections** — the headings you already wrote, plus any region you mark yourself — and never on screen coordinates. There is no cookie, no fingerprint, no visitor id, and nothing that survives closing a browser tab.

> **Content Reach is available on the Business and Enterprise plans**, and needs the AI assistant configured (it shares that gateway key). It is **off by default** even then — turn it on deliberately.

Turn it on

```
contentReach:
  enabled: true
```

Rebuild, and the Content Reach tab in your Aardvark cloud dashboard starts filling in. The full set of options:

Key	Default	What it does
`enabled`	`false`	Turns the whole feature on.
`requireConsent`	`false`	Wait for an explicit consent signal before recording anything. See Consent.
`sampleRate`	`1.0`	Record this fraction of browser tabs (`0.0`–`1.0`). The draw happens once per tab, so a sampled reader's whole journey is kept together.
`targetsOnly`	`false`	Track only the regions you mark yourself, skipping the automatic heading sections.

What gets measured

Five things, and it is worth being precise about each — the numbers are only useful if you know exactly what they count.

Visible time

How long the page was in the **foreground** of a visible tab. Time stops when the reader switches tabs or minimizes the window, and stops again after 60 seconds with no scroll, click, keypress, or focus. Capped at one hour per page view.

Sections reached

A section counts as **reached** when it is at least half visible for at least one continuous second. “Half visible” means either half of the section, or half of the screen — the second case matters, because a section longer than the viewport can never be half of itself while filling the reader’s entire screen.

Sections inside a collapsed accordion or an inactive tab are never reached, which is correct: nobody could see them.

Section dwell

How long each section spent as the one the reader was positioned on — the same “you are here” rule that highlights an entry in the **On this page** sidebar. Exactly one section is current at a time, so the per-section times always add up to no more than the page’s visible time. Capped at ten minutes per section per page view, and the dashboard reports the **median** rather than the average so one tab left open overnight can’t define a section’s number.

Time spent above the first heading is credited to a synthetic **intro** section, so a page’s opening paragraphs aren’t silently attributed to the first heading below them.

By default Content Reach measures **h2 and h3** anchors (the same band the table of contents uses). If you enable `search.sections` with a wider or narrower heading band, Reach uses that band too — so the heat strip denominator agrees with the anchors search indexes. Pages with more than 40 sections measure only the first 40 in document order (matching the per-beacon wire cap); the completion denominator is that measured set, so a reader who covers every tracked band can still reach 100%.

A section can be **reached with zero dwell**, and that isn’t a bug — the two measures answer different questions. A long section can fill a reader’s screen (so it is reached) while its heading is still below the top of the viewport, in which case the reader is still positioned in the section above it and the time belongs there. Expect the sections a reader stopped short of to show reach without dwell.

Once a reader is at the bottom of the page, the time goes to the **last** section, since that is what they are looking at and what the On this page sidebar highlights. On a short page whose final sections all fit on screen at once, that means the last one absorbs the tail of the visit. Dwell is most informative on pages long enough that sections scroll past one at a time.

Last section reached

The furthest section each page view got to. This is a **proxy**, and the dashboard labels it as one: a reader who jumps to the bottom of the page from the table of contents sets it exactly like one who read all the way down. It is where readers got to, not where they gave up.

Deliberate actions

Five fixed kinds, counted per section:

- **Copy code** — the copy button on a code block was pressed. Never what was copied.
- **CTA clicks** — a link or button inside a region you marked.
- **TOC jumps** — an "On this page" link was clicked, counted against the section it points to.
- **Search arrivals** — the reader landed on this section from your site's search box. No query text crosses over.
- **Expansions** — an accordion or tab control in this section was used.

The list is closed. Adding a sixth kind takes a new Aardvark release; nothing your pages contain can invent one.

What it is not

Content Reach measures **visibility and deliberate interaction**. That is all it can measure, and the wording throughout the product reflects it:

- A section being on screen is not evidence anyone **read** it — a reader can scroll past with their eyes closed.
- Dwell time is not **attention**. Someone can leave a section on screen while doing something else; the idle cutoff removes the worst of that, not all of it.
- Nothing here measures **comprehension**, **gaze**, or **emotion**, and no combination of these numbers can be turned into such a claim.
- "Last section reached" is not a **drop-off point**.

If you need to know whether people understood something, ask them — reader surveys and page ratings are the tools for that.

Privacy

The design constraint is that Content Reach must be useful without knowing anything about who anyone is.

Never collected: cursor or scroll coordinates, screen recordings, page snapshots, element or clipboard text, form values, search terms, referrers, user agents, IP-derived data, or any identifier that outlives a browser tab.

Collected: the page path, the ids of the sections that came into view, how long each held the reader's position, and counts for the five actions above.

Readers are grouped by a random **tab session id** held in `sessionStorage`. It is minted per tab, dies when the tab closes, and is deliberately separate from the one search analytics uses, so no identifier links a Content Reach session to a search session. There is no "unique visitors" number anywhere in the dashboard, because there is nothing to count.

No query text ever reaches Content Reach — a search arrival hands over the fact that it happened and nothing else. What the two do share is the page path and roughly the time, because both are

per-page analytics and neither is useful without them. On a quiet page, someone who can read both datasets could therefore line a search up with a Content Reach session even though nothing joins them. If that inference matters for your site, keep the terms out of storage with `search.analytics.store_terms: false` — the funnel still works, and there is then no query text on either side to line up.

Readers with **Do Not Track** or **Save-Data** enabled send nothing at all.

Data is deleted after **90 days** by default. Nothing derived outlives it — there are no long-lived summaries that keep a copy after the raw rows expire.

Consent

Content Reach uses your site's **existing** consent signal; Aardvark ships no cookie banner of its own. With `requireConsent: true`, nothing is recorded until your banner sets `window.aardvarkConsent` and dispatches an event:

```
window.aardvarkConsent = true;
window.dispatchEvent(new Event('aardvark-consent'));
```

This is the same contract reader surveys use. Set `aardvarkConsent` to a function if the answer can change. Until then the gate **fails closed** — no signal means no recording — and a reader who accepts partway down a page is measured from that moment on, never retroactively.

Whether you need this depends on your jurisdiction and your own legal advice. Content Reach sets no cookies and stores no personal data, but "no consent needed" is not a conclusion Aardvark can reach on your behalf. Wire up a banner with `integrations.custom` and turn `requireConsent` on if in doubt.

Mark a high-value region

Headings become sections automatically. For the blocks that matter most but aren't headings — an install command, an auth step, a "Try it now" — wrap them in `{% target %}`:

````
{% target id="install-cli" %}
```bash
npm install -g aardvark
```
{% endTarget %}
````

The id is a lowercase slug, in the same namespace as a heading anchor, so the dashboard shows one ordered list of sections rather than two. A malformed id warns at build time and renders the content unwrapped — you lose the measurement, never the content.

For a single component instead of a region, no new syntax is needed:

```
{% button url="/signup" attr={"data-aardvark-target": "signup-cta"} %}Start free{% endButton %}
```

Section discovery is a single snapshot taken once the page has mounted, so the marked element has to exist by then. That holds for `{% target %}` and for any component that renders a real element — but a handful of Mantine components apply `attr` several frames after mounting, and a target on one of those can miss the snapshot and go unmeasured. Nothing breaks and no warning fires; the region simply never appears in the dashboard. If a marked region doesn't show up, wrap it in `{% target %}` instead of marking the component.

Full reference: Target.

Reading the dashboard

The **Content Reach** tab opens on the aggregate: page views, tab sessions, average visible time, and average sections reached, followed by a table of pages sorted by traffic with a reach-completion bar. Export it as CSV from the header.

Click any page for its **section heat strip** — one band per section in document order, shaded by either reach or median dwell, with the action counts beside it. Every band prints its own number and carries a screen-reader label, so the strip reads correctly in greyscale and with assistive technology.

If you rename or remove a heading, its recorded history stays for the rest of the retention window — those numbers really happened. The strip keeps showing them, but they are counted separately and called out as no longer part of the page, so a heading you renamed last week never reads as a section of today's page that nobody reached.

It is a section strip rather than a pixel heatmap by design. Aardvark knows which section of your page a reader reached; it deliberately does not know where anything was on their screen, and a pixel map would require exactly the coordinate tracking this feature refuses to do.

Turning it off

Remove the `contentReach` block (or set `enabled: false`) and rebuild. The client stops shipping with your site immediately, and existing data ages out on the normal 90-day window.

Analytics & page ratings

Aardvark ships native support for the same analytics platforms as Mintlify. Add a block under `integrations:` with your platform's id or key and Aardvark injects that vendor's official `<script>` on every page — no template editing. Platform keys and legacy fields remain Mintlify-compatible, while vendor-current additions are available too. In particular, Plausible now prefers its per-site `scriptId`; an existing Mintlify-style `domain` block continues to work unchanged.

```yaml
integrations:
  ga4:
    measurementId: "G-XXXXXXXXXX"     # Google Analytics 4
  plausible:
    scriptId: "pa-XXXXXXXXXXXX"       # copy from Plausible's Site Installation snippet
  posthog:
    apiKey: "phc_XXXXXXXX"
```

A block whose required field is empty (e.g. `measurementId: ""`) is treated as off — leave it empty to keep that platform silent, and set it to your real id to switch it on. (This sample site wires a real GA4 id so its live demo records analytics — see `integrations.analytics` in `aardvark.config.yaml`.) If you set a block but misspell its required field, the build prints a one-line warning so the mistake isn't silent.

> **Not a secret.** Every analytics id ships in the page's HTML by design, so committing it is fine. For per-deploy values, a build-time env var is still cleaner — see the `AARDVARK_KEY` note under the AI assistant for that pattern.

Supported platforms

Each platform is a block under `integrations:`. Set the **required** field(s) and you're done.

Platform	Config key	Required	Notes
Google Analytics 4	`ga4`	`measurementId`	Also accepts the legacy `analytics.measurementId` key
Google Tag Manager	`gtm`	`tagId`	Injects the head loader **and** the `<noscript>` fallback
Plausible	`plausible`	`scriptId` (`id` alias)	Preferred current per-site tracker; optional `scriptSrc` replaces only its loader URL. Legacy `domain` + optional `src` remains supported unchanged

Fathom	`fathom`	`siteId`	
PostHog	`posthog`	`apiKey`	Optional `apiHost` (default `https://us.i.posthog.com`; use `https://eu.i.posthog.com` or a proxy)
Mixpanel	`mixpanel`	`projectToken`	Tracks pageviews by default. Session Replay and heatmaps ship on by default (the vendor's current recommendation) — optional `recordSessionsPercent` (integer 0–100, default `100` = every session) caps how much replay is recorded (use `0` to record no sessions) and `recordHeatmapData` (`true`/`false`, default `true`) toggles heatmaps; setting either empty/invalid warns and turns just that feature off
Amplitude	`amplitude`	`apiKey`	Browser SDK 2 with autocapture + remote config; network-request capture is left **off** so request URLs/bodies aren't sent to Amplitude. Session Replay ships on by default — optional `sessionReplaySampleRate` (0–1, default `1` = every session) caps how much is recorded (use `0` to load the plugin but record nothing; omit the key for the full-rate default — leaving it empty/`null` warns rather than recording everything)
Microsoft Clarity	`clarity`	`projectId`	
Heap	`heap`	`appId`	
Hotjar	`hotjar`	`hjid`, `hjsv`	`hjsv` is the snippet version (defaults to `6`)
LogRocket	`logrocket`	`appId`	
Pirsch	`pirsch`	`id`	The site's identification code
Segment	`segment`	`key`	The source write key; emits Segment's current browser snippet protocol (`5.2.1`)
Clearbit	`clearbit`	`publicApiKey`	Now HubSpot Breeze Intelligence
Hightouch	`hightouch`	`writeKey`	Optional `apiHost` (default `us-east-1.hightouch-events.com`)

Adobe Analytics	`adobe`	`launchUrl`	The full Adobe Launch / AEP Tags embed URL

A fuller example mixing several:

```yaml
integrations:
  fathom:
    siteId: "ABCDEFGH"
  hotjar:
    hjid: 1234567
    hjsv: 6
  segment:
    key: "wk_live_XXXXXXXX"
  posthog:
    apiKey: "phc_XXXXXXXX"
    apiHost: "https://eu.i.posthog.com"   # optional override
  plausible:
    scriptId: "pa-XXXXXXXXXXXX"
    scriptSrc: "/js/plausible.js"   # optional custom loader URL
```

Plausible's current snippet uses a unique script id for each site. Copy the id from the `https://plausible.io/js/<scriptId>.js` URL shown under **Site Installation**; Aardvark emits that async loader plus Plausible's queue and `plausible.init()` shim. `id` is accepted as a short alias. If both `scriptId` and `domain` are present during a migration, `scriptId` wins. The current mode's override is named `scriptSrc` deliberately: legacy `src` commonly points at the old `script.js`, which cannot replace the new per-site loader. A legacy `src` alongside `scriptId` is ignored with a build warning so a partial migration cannot silently mix protocols. `scriptSrc` changes only where the JavaScript is loaded from; it does not change Plausible's event endpoint, so it must not be treated as complete proxy configuration.

Existing sites do not have to migrate immediately. The older domain-based form remains a legacy compatibility path and keeps emitting the previous `defer`/`data-domain` snippet:

```yaml
integrations:
  plausible:
    domain: "docs.example.com"
    src: "https://plausible.example.com/js/script.js"   # optional legacy self-host
```

> **Migrating from Mintlify?** Copy the `analytics`/integration entries from your `docs.json` straight into `integrations:` (YAML instead of JSON). The legacy keys and fields stay compatible, with two name fixes Mintlify itself uses: Segment's field is `key`, Hightouch's is `writeKey`. A Plausible `domain` block still works; switch it to the per-site `scriptId` when you are ready to adopt Plausible's current tracker. Mintlify's `cookies` and `telemetry` entries aren't analytics platforms — `telemetry` is Mintlify-internal, and for a cookie-consent banner use a custom snippet.

Which parts of a page readers reach

The events above tell you a page was opened and how it was rated. To see which sections of it readers actually got to — and which install commands were copied, which CTAs were clicked — turn on Content Reach (Business and Enterprise). It records section visibility using your page's own heading anchors, with no cookies, no coordinates, and no visitor identity.

Anything else? Inject any snippet

For a tracker not in the table above — or any other third-party tag — paste its snippet into `integrations.custom`. The `head` value is emitted verbatim into every page's `<head>`, and `body` just before the closing `</body>`:

```
integrations:
  custom:
    head: |
      <script defer src="https://cdn.example-analytics.com/tag.js"
              data-site="my-site"></script>
    body: |
      <noscript><img src="https://example-analytics.com/noscript.gif"></noscript>
```

This content is injected **raw** (it isn't escaped — that's the point), so treat it like the theme template: only put trusted markup there. It's injected **after** the built-in integrations (the analytics platforms above), so a `custom` snippet can reference them.

Google Analytics

GA4 is the one platform with deeper integration: the page-rating and reader-survey widgets below record their results as `gtag` events, so they light up automatically once GA is configured (under either `ga4.measurementId` or the legacy `analytics.measurementId`).

```
integrations:
  ga4:
    measurementId: "G-XXXXXXXXXX"
```

Aardvark injects the standard `gtag.js` snippet. An empty id disables it; this sample site sets a real id, so GA is active here.

Page ratings

The default theme adds a **"Was this page helpful?"** widget at the bottom of every page — the built-in `PageFeedback` island. A visitor picks a star rating (they can adjust it — the value they settle on is what's recorded); the widget thanks them and reveals an optional comment box (a Mantine `Textarea` + a **Submit** button). Two Google Analytics events fire, each once per page:

```
// once, after the reader settles on a star (re-picks within ~600ms are debounced)
gtag('event', 'page_rating', {
  event_label: location.pathname,   // which page was rated
```

```
  value: 4,                          // the chosen number of stars (1-5)
});

// when the optional comment is submitted
gtag('event', 'page_rating_comment', {
  event_label: location.pathname,                // which page the comment is about
  comment: 'the search box was hard to find',    // capped at ~100 chars
  value: 4,                                       // the star rating they gave
});
```

The widget always renders; the events only fire when Analytics is configured (so `gtag` exists). The page URL rides in its own `event_label` param so it's always captured in full, and the comment is a separate `comment` param — GA4 caps each parameter value at ~100 characters, so the box stops accepting input at 100 (with a live counter) and the value is trimmed defensively before sending.

The reader can change their pick before it's recorded — re-picks are debounced, so a 3 → 5 correction sends a single `page_rating` with the final 5; once recorded, the stars lock so the visible rating always matches what was sent.

To customize the wiring — send the events elsewhere (your own endpoint, PostHog, …), change the copy, or restyle the box — drop a `snippets/PageFeedback.jsx` into your project; a project snippet of that name overrides the built-in island. To drop the widget entirely, set `pageFeedback: false` in `aardvark.config.yaml`. (The standalone `{% rating %}` component is still available for star ratings anywhere in your content.)

> **Privacy:** the comment is free-form text sent to Google Analytics, so the widget shows a "Please don't include personal information." notice under the box by default (and comments only leave the browser when Analytics is configured). If you operate under GDPR/CCPA or similar and need an explicit disclosure, a privacy-policy link, or consent before capture, change the `commentNotice` string or gate the widget via `snippets/PageFeedback.jsx`.

> **GA4 setup:** `comment` and `event_label` are custom event parameters — GA4 collects them (visible in DebugView and the BigQuery export) but won't show them in standard reports or Explore until you register each as a **custom dimension** (Admin → Custom definitions → Custom dimensions, scope: **Event**, parameter names `comment` and `event_label`). Without that, comments are still captured but never appear in the GA4 UI. `value` is built in and needs no setup.

> **Note:** GA4 treats `value` as its built-in (monetary) metric. We reuse it for the star count — it records and averages fine in reports, but if that collides with your own use of `value`, rename it to a custom parameter in `snippets/PageFeedback.jsx`.

> **Upgrading?** The page-feedback widget changed from a / pair to a 1–5 star rating, and the `page_rating` event payload changed with it — the old `{ rating: 'up'|'down', value: 1|0, page_path }` is now `{ event_label: <path>, value: 1–5 }`. Update any GA dashboards, custom dimensions, or alerts that filtered on `rating`, `value == 1`, or `page_path`. The `page_rating_comment` event is new.

Reader surveys

A reader **survey** is a small inline prompt — modeled on the discontinued Google Surveys product — that asks site visitors a short question and records their answers to **Google Analytics**. It appears at the end of every page and asks one question at a time.

It's off by default. Turn it on with a `survey:` block:

```yaml
survey:
  enabled: true
  title: "Quick question"     # the card heading
  sampleRate: 1.0             # fraction of visitors who ever see it (1.0 = everyone)
  questions:
    - type: single            # multiple choice — pick one
      text: "What brought you here?"
      choices: ["Reference", "Tutorial", "Troubleshooting"]
      randomize: true         # shuffle the answer order
    - type: rating            # a 1..max star scale
      text: "How easy was it to find what you needed?"
      max: 5
    - type: multi             # checkboxes — pick several
      text: "Which sections have you used?"
      choices: ["Guides", "API", "Changelog"]
      noneOfTheAbove: true    # adds a pinned, exclusive "None of the above"
    - type: text              # open-ended; capped at 100 characters
      text: "Anything we could improve?"
      placeholder: "Up to 100 characters"
```

Survey options

These keys go directly under `survey:`.

Option	Default	What it does
`enabled`	`false`	Turns the survey on. It's opt-in, so it stays off until this is truthy.
`questions`	—	The list of questions, asked one at a time — see **Question options** below. At least one valid question is required, or the prompt doesn't render.
`title`	`Quick question`	The card heading when the survey asks **more than one** question. With a single question, only the question's `text` is shown (the heading would duplicate it).
`sampleRate`	`1.0`	Fraction of visitors (`0`–`1`) who ever see it.

`requireConsent`	`false`	Gate the prompt and its GA events behind analytics consent (see **Consent gating** below).
`id`	content hash	The GA `survey_id` label. Defaults to a hash of the questions; set it explicitly for a stable label when you edit the wording (see **Survey identity** below).

Question types

`type`	**Renders**	**Recorded to GA as**
`single`	radio group (pick one)	the chosen label
`multi`	checkbox group (pick several); `noneOfTheAbove: true` adds a pinned, exclusive option	the chosen labels, comma-joined
`rating`	a 1–`max` star scale (default 5)	a numeric `value` (1–`max`)
`text`	a short textarea	the typed text, capped at 100 characters in GA (the full text is also stored in the gateway when the AI assistant is on)

Question options

Each entry in `questions` is a mapping. `type` and `text` apply to every question; the rest depend on the type (an option set on a type that ignores it is simply unused).

Option	**Applies to**	**Default**	**What it does**
`type`	all	`single`	The control to render: `single`, `multi`, `text`, or `rating` (see **Question types** above).
`text`	all	— (required)	The question shown to the reader. A question with no `text` is dropped with a build warning rather than rendered broken.
`choices`	`single`, `multi`	— (required)	The answer options, as a list of strings. Each label is capped at 100 characters. A `single`/`multi` question with no `choices` is dropped (with a warning).

`randomize`	`single`, `multi`	`false`	Shuffle the answer order each time it's shown; a `noneOfTheAbove` option stays pinned last.
`noneOfTheAbove`	`multi`	`false`	Add a pinned, exclusive "None of the above" checkbox — picking it clears the other selections, and picking a real option clears it.
`required`	all	`false`	Require an answer before **Submit** is enabled for that question (otherwise it can be left blank and skipped).
`max`	`rating`	`5`	The number of stars; the scale runs `1`–`max` (minimum 2).
`placeholder`	`text`	—	Hint text shown in the empty textarea.
`maxLength`	`text`	`100` (`2000` with the assistant on)	Maximum characters in the box (a live counter shows usage; the field stops input at the limit). Clamped to 100 — GA4's per-parameter limit — normally. When the AI assistant is enabled the ceiling rises to 2000 (the gateway stores the full comment; GA still keeps the first 100), and an unset `maxLength` defaults to 2000. A smaller value you set is honored either way.
`id`	all	`q1`, `q2`, …	The GA `question_id` for this question. Defaults to the question's surviving position; set an explicit value for a stable label that won't shift if you reorder or add questions.

Invalid config

At build time each question is validated before the prompt renders. Broken entries are **dropped with a warning** rather than shown to readers — the rest of the survey still runs, and surviving questions are renumbered (`q1`, `q2`, …) so GA `question_id` values have no gaps. If nothing valid remains, the survey prompt doesn't render at all.

Problem	What happens
Missing or blank `text:`	Question dropped (warning names its position in the list).
`single` / `multi` with no `choices:`	Question dropped.

Unknown `type:`	Question dropped. Valid types: `single`, `multi`, `text`, `rating`.

Common synonyms are accepted silently and mapped to the canonical type:

You write	Maps to
`textarea`, `open`, `comment`	`text`
`select`, `radio`	`single`
`checkbox`, `checkboxes`	`multi`
`stars`, `scale`	`rating`

Omitting `type:` defaults to `single` — you only need to set it when you want something else.

When it shows

The prompt appears on **every page**, as soon as the reader is eligible — there's no "browse a few pages first" delay.

- **Per-page opt-out.** Set `survey: false` in a page's front matter to suppress the prompt on that page while keeping it enabled site-wide. A page that says nothing inherits the site setting. Surveys must be enabled in `aardvark.config.yaml` first — front matter cannot turn them on for a site where they are off.
- **Sampling.** `sampleRate` (0–1, default `1.0`) shows it to only a stable fraction of visitors: one roll is stored per visitor, so the same person is consistently in or out across pages.

Survey identity

Every GA event carries a `survey_id`. By default it's a short content hash of what's asked (question type, wording, choices, rating scale) — so editing the substance creates a new id in reports. Presentation-only changes (`title`, `randomize`, `noneOfTheAbove`, `required`, per-question `id`, etc.) do **not** change the hash. Set an explicit top-level `id` on the `survey` block when you want a stable GA label across edits.

Completing the survey on one page does **not** hide it on the next — the prompt is meant for page-specific feedback (`response_page_url` in each event identifies where the answer was left). Use per-page `survey: false` front matter to omit it from pages where it doesn't belong.

What gets recorded

Answers ride `gtag` events, so they're only recorded when Analytics is configured (and `gtag` therefore exists):

```
gtag('event', 'survey_impression', { survey_id });                          // first shown
gtag('event', 'survey_response',   { survey_id, question_id, answer, response_page_url:
window.location.pathname.slice(0, 100) });    // per answer
gtag('event', 'survey_response',   { survey_id, question_id, value: 4, answer: '4',
response_page_url: window.location.pathname.slice(0, 100) }); // rating
gtag('event', 'survey_complete',   { survey_id });                           // finished
```

`answer` is always capped at 100 characters (GA4's per-parameter limit): open-ended text stops at 100 with a live counter, and a multi-select joins whole labels up to that cap.

> **Privacy:** open-ended (`text`) answers are free-form text sent to Google Analytics, which prohibits storing personally identifying information. The box shows a "Please don't include personal information." notice by default; prefer `single` / `multi` / `rating` where you can, and review your GA data-retention settings. Under GDPR/CCPA, set `requireConsent` to gate the prompt and its events behind your consent flow — see **Consent gating** below.

> **GA4 setup:** `survey_id`, `question_id`, `answer`, and `response_page_url` are custom event parameters — GA4 collects them (visible in DebugView and the BigQuery export) but won't show them in standard reports until you register each as a **custom dimension** (Admin → Custom definitions, scope: **Event**). `value` is GA4's built-in numeric metric and needs no setup. `response_page_url` is `window.location.pathname` — the path only, no origin or query string (avoids PII in query params), capped at 100 characters.

Full comments with the AI assistant

GA4 caps every event parameter at 100 characters, which truncates a real comment. When your site has the AI assistant enabled (`ai.assistant`), the survey **also** posts each open-ended **`text`** answer to your Aardvark **gateway** at full length — up to **2000 characters** — so the whole comment survives. This is automatic; there's nothing extra to configure beyond enabling the assistant.

- **What's sent:** the comment text (full), the page it was left on (`page_url`, pathname only — same path-only, no-query rule as the GA event), and the `survey_id` / `question_id`. It rides the same baked public key and origin allowlist as the assistant's chat and feedback calls, so no new credentials are involved. It's best-effort and never blocks the reader.
- **What still goes to GA:** everything above — the `survey_response` event still fires with the first 100 characters of the answer and the page. GA keeps the aggregate view; the gateway keeps the full text.
- **Where to read them:** open your gateway **dashboard**, paste your secret key, and scroll to **Page comments** — the most recent 50, each with its page and full comment text.
- **Scope:** only `text` answers go to the gateway (that's where the 100-char cap is lossy). `single` / `multi` / `rating` answers are short and stay GA-only. With the assistant **off**, the survey behaves exactly as documented above — GA only, `text` capped at 100.

> **Privacy:** the same caution as the GA path applies, and more so since the gateway keeps the full text — readers see the "Please don't include personal information." notice, but review your

> gateway's data handling and retention before enabling it on sensitive sites.

Consent gating

By default the survey shows and records as soon as a reader is eligible. Where you need consent before any analytics — GDPR/CCPA, say — set `requireConsent: true`: until the page signals that analytics consent was granted, the prompt stays hidden, **no** gtag event fires, and **nothing** is written to local storage (the sampling roll only).

The signal is a global your consent banner controls — a boolean, or a function for live state:

```
// after the reader accepts analytics:
window.aardvarkConsent = true;
// …or a function, to read a consent manager on demand:
window.aardvarkConsent = () => myCMP.hasConsent('analytics');

// then tell a survey that's already on the page to re-check — no reload needed:
window.dispatchEvent(new Event('aardvark-consent'));
```

The gate **fails closed**: with `requireConsent: true` and no clear `true` signal, the prompt never shows. Withdrawing consent (set it falsy and dispatch the event again) hides a visible prompt and resets it, so a later re-grant reopens it fresh at the first question. Since gating the prompt also gates every event, nothing reaches GA until consent is granted — point `aardvarkConsent` at the same state your site's `gtag` Consent Mode already manages.

> **Impression counting:** `survey_impression` fires once each time the prompt is shown — normally once per visit, but a withdraw-then-re-grant toggle reopens it (fresh, from question 1) and so records another impression. If you compute a response rate as responses ÷ impressions, note that a consent toggle can push impressions above one per visitor.

Accessibility

The prompt is an inline card, never a focus-trapping modal. Every control is a standard labelled input and a **Submit** button; as the survey advances, the new question — and the closing thank-you — are announced through an `aria-live` region. See the Accessibility page.

Customizing

The survey is the built-in `Survey` island. To change the flow, restyle the card, or send answers somewhere other than Google Analytics (your own endpoint, PostHog, …), drop a `snippets/Survey.jsx` into your project — a project snippet of that name overrides the built-in island. To turn the prompt off site-wide, set `survey.enabled: false` in `aardvark.config.yaml` (or `survey: false` to disable the whole block). To hide it on individual pages, set `survey: false` in that page's front matter.

Definitions & glossary

`definitions.yaml` is Aardvark's **translation memory** — the single source of truth for how terms, acronyms, and product names are handled across languages. Every entry is injected as a grounding instruction whenever `vark build --translate` runs, so the model never guesses how to render your vocabulary.

Entries marked `public: true` do double duty: they also power a **reader-facing glossary**, adding a dotted-underline hover card to the first mention of each term on every page, and optionally generating a standalone glossary page.

Setup

Point `aardvark.config.yaml` at your file:

```
definitions: definitions.yaml
```

The path is relative to your project root. The file is optional — if it's absent, both features are silently skipped.

File format

```
terms:
  - term: island
    definition: A self-contained interactive React component rendered in the browser.
    public: true

  - term: Static Site Generator
    acronym: SSG
    definition: A tool that turns source files into static HTML pages.
    public: true

  - term: build
    definition: The process that turns Markdown sources into static HTML output.
```

Each entry supports:

Field	Required	Default	Description
`term`	yes	—	The canonical term (case-preserved).
`definition`	no	`""`	Plain-text definition. For public terms: shown in the hover card and on the glossary page (a public term with no definition emits a build warning). For all terms: serves as a context note in the translation prompt when no exact translation is provided.

`acronym`	no	—	Optional acronym. The hover card title becomes “Term (ACRONYM)”.
`public`	no	`false`	Show a hover card on first mention + list on the optional glossary page.
`translate`	no	`true`	Set `false` to keep the term verbatim in every language (product names, trademarks).
`translations`	no	—	Per-language overrides (see below).

Per-language overrides

Use `translations:` to supply translated terms, acronyms, or definitions for specific languages:

```yaml
terms:
  - term: Static Site Generator
    acronym: SSG
    definition: A tool that turns source files into static HTML pages.
    public: true
    translations:
      fr:
        term: générateur de site statique
        acronym: GSS
        definition: Un outil qui transforme des fichiers source en pages HTML statiques.

  - term: Markdown
    definition: The lightweight markup language these docs are authored in.
    public: true
    translate: false          # keep "Markdown" verbatim in every language

  - term: build
    definition: The process that turns Markdown sources into static HTML output.
    translations:
      fr: compilation         # string shorthand — equivalent to {term: compilation}
```

A `translations.<code>` value can be:

- **A string** — shorthand for `{term: <string>}`. Useful for simple term substitutions.
- **A mapping** — any of `term`, `acronym`, `definition`. Omitted keys fall back to the base values.
- **`false`** — keep the base term and acronym verbatim for that language (same as `translate: false` but per-language). Also accepted as `{do_not_translate: true}`.

Reader-facing hover cards

For every entry with `public: true`, Aardvark wraps the **first mention** of the term on each page in a dotted-underline span. Hovering (or tabbing to) the term shows a card with the title and definition.

A few rules govern which text is decorated:

- **First occurrence only** — both the term form and the acronym form each get decorated once per page independently, so "SSG" gets a card and "Static Site Generator" gets a card.
- **Headings are skipped** — a dotted underline in an `<h2>` is visual noise.
- **Link text is skipped** — the term's underline would conflict with the link's underline.
- **Code is never touched** — inline code and fenced code blocks are their own token type and are never scanned.
- **Matching** — terms match case-insensitively; acronyms match case-sensitively (so `SSG` matches but `ssg` does not). Longer surfaces win over shorter ones they contain. Word boundaries are enforced ("`island`" never matches inside "`islander`").

The card title is "Term (ACRONYM)" when an acronym is present, otherwise just "Term". On non-English pages, the localized term, acronym, and definition are used — a French page shows the French card.

Keyboard access

Tab to a glossary term to reveal the card immediately. Press Escape to close it without moving focus. This satisfies WCAG 1.4.13 (content on hover or focus).

Optional standalone glossary page

To generate a standalone glossary page, add a `page:` block to `definitions.yaml`:

```yaml
page:
  enabled: true
  url: /definitions/glossary/
  frontmatter:
    title: Glossary
    menu: docs
    weight: 57
```

The page is generated for each content language your site has. A French site also gets `/fr/definitions/glossary/` with the French terms. The page body lists all public terms alphabetically as `## Term (ACRONYM)` headings followed by their definitions — so each term gets an anchor and a TOC entry.

The `frontmatter:` block accepts the same front matter any content page uses: `title`, `menu`, `parent`, `weight`, `description`, `icon`, and so on. Set `menu` + `parent` to slot the page into your navigation.

A few safety checks apply:

- `url` must be a non-root path (not `/`) — the homepage can never be overwritten.
- `url` must contain no `.` or `..` segments.
- If a content page already occupies the URL, the generated page is skipped with a warning.
- The glossary page itself is never self-decorated — its term list doesn't get hover cards.

Translation memory

`definitions.yaml` is Aardvark's translation memory. Every entry — not just public ones — is compiled into a set of grounding instructions that prefix the model's prompt for each page translation. This is what keeps your domain vocabulary consistent: the model follows explicit rules rather than re-deriving terminology page by page.

The prompt lines Aardvark generates depend on what each entry has:

Entry has	Prompt instruction
`translations.fr.term` (exact translation)	`"Static Site Generator" must be translated as "générateur de site statique" (A tool that turns source files into static HTML pages.)`
`translations.fr.acronym` (different acronym)	`render the acronym "SSG" as "GSS"`
No translation, but a `definition`	`"island": A self-contained interactive React component rendered in the browser.`
`translate: false` (no acronym)	`"Markdown" must NOT be translated; keep it verbatim (The lightweight markup language...)`
`translate: false` + `acronym`	Two lines: the verbatim-keep instruction above, plus `keep the acronym "ADV" verbatim`
Per-language `false` or `do_not_translate: true`	Same verbatim-keep instruction(s), for that language only

What to put in the file:

- Terms with exact target-language translations: supply `translations.<code>.term` (and `acronym` if it differs). The model is told exactly what to write.
- Terms without translations but with a `definition`: the definition serves as a context note. The model can freely translate the term but understands what it means — useful for common domain vocabulary.

- Product names, brand names, and trademarks: set `translate: false` (all languages) or `translations.<code>: false` (one language). The model keeps them verbatim.
- Acronyms that expand differently in each language: use `translations.<code>.acronym`.

Non-public entries ground translation just as well as public ones. The `public` flag only controls the reader-facing hover card and glossary page — it has no effect on translation.

```
# Translate missing/changed pages (needs AARDVARK_SECRET_KEY):
vark build --translate

# Re-translate everything, overwriting existing translations:
vark build --retranslate-all
```

The multi-language setup (source directories, language codes) is configured under `languages:` in `aardvark.config.yaml`. `definitions.yaml` is the terminology layer on top of that — it says how to handle specific words, not what languages the site has.

Migrating from `languages.translationMemory`

The old `languages.translationMemory` key is no longer read. If you have it in your config, move it to a top-level `definitions:` key. The file format has also changed: instead of a plain term→translation map, entries are now full objects with optional acronym, definition, and `public` flag.

```
# Before (no longer works):
languages:
  translationMemory: translation-memory.yaml

# After:
definitions: definitions.yaml
```

A build warning is emitted while the old key is still present so you don't silently lose translation grounding.

Accessibility

Every site Aardvark generates targets **WCAG 2.1 Level AA** out of the box. The chrome ships with semantic landmarks, a skip link, visible keyboard focus, screen-reader announcements, and reduced-motion support — and the build flags color-contrast problems in your theme before they reach readers.

Standards we follow

- **WCAG 2.1 Level AA** — the contrast, keyboard, and name/role/value success criteria.
- **Keyboard-first** — every interactive control is reachable and operable without a mouse.
- **Screen-reader friendly** — landmarks, current-state cues, and a live region for dynamic updates.
- **Respects user preferences** — `prefers-reduced-motion` and `prefers-color-scheme`.

Built-in accessibility features

- **Skip to content** — the first `Tab` stop on every page jumps past the header and navigation straight to the main content.
- **Landmark regions** — the banner, the section tabs, the documentation sidebar, the breadcrumb trail, the main content, and the "On this page" list are each exposed as a labelled region, so screen-reader users can jump between them.
- **Visible focus** — every link, button, and control shows a clear focus ring when you navigate with the keyboard.
- **State cues** — the current page (sidebar), the current section (on-this-page), the in-view API operation (sidebar), expanded/collapsed sections, and sorted table columns are announced, not just shown with color.
- **Live announcements** — filtering a table announces its result ("No matching rows") to assistive technology.
- **API response examples** — the API reference shows a response's example body as a collapsible JSON tree; its expand/collapse-all and copy controls are standard focusable buttons, so a keyboard user can expand and copy the whole body without a mouse. No new keyboard shortcut is introduced, and without the optional viewer it falls back to a plain code block.
- **Links beyond color** — body links are underlined, so they're distinguishable without relying on color.
- **Reduced motion** — scroll-spy, tab, and reveal animations are disabled when your system asks to reduce motion.
- **Password unlock form** — a password-protected page's unlock prompt is a real labelled `<form>`: it's fully keyboard-operable (the field autofocuses, `Enter` submits), shows a visible focus ring, and announces a wrong-password message through an `aria-live` region. No new keyboard

shortcut is introduced.

- **Taxonomy listing filters** — a taxonomy listing's tag badges are keyboard toggles (`Enter`/`Space`, with `aria-pressed` state), a knowledge base's category cards are real buttons with the same pressed cue, and the knowledge base announces an active filter's result count through a status region. Filter state mirrors into the URL hash so a filtered view can be linked and shared; knowledge-base category picks push a history entry, so `Back` restores the previous filter state there. No new keyboard shortcut is introduced.

Keyboard shortcuts

The chrome and the interactive components are fully keyboard-operable:

Where	Keys	Action
Anywhere (on load)	`Tab`	Focus **Skip to main content**; `Enter` jumps to the article
Sidebar navigation	→ / ←	Expand / collapse the focused section
Sidebar section toggle	`Enter` / `Space`	Open or close the section
Section tab menu (header)	`Esc`	Close the dropdown and return focus to its button
Search dialog	type · ↑ / ↓ · `Enter` · `Esc`	Search the index, move through visible results, open the focused result, or close the dialog
Search path filter	`Tab` · type · `Enter` / `Space` · `Backspace` · `Esc`	Reach the path filter, search path options, select or deselect paths, remove a selected path, or close the dropdown
Search exact matches	`Tab` · `Space`	Toggle exact matching; focus the help icon to read what changes
Table column header	`Enter` / `Space`	Sort by that column
Table filter box	type to filter · `Esc`	Show only matching rows (live), or clear and dismiss the filter
Tabbed content	← → `Home` `End`	Move between / jump to first or last tab

Accordion / API sections	`Enter` / `Space`	Expand or collapse
File tree	↑ ↓ → ← `Home` `End`	Move focus; expand/collapse a folder or step in/out; jump to first/last row
File tree — activate row	`Enter` / `Space` · `Esc`	Toggle a folder, open a file's code in a modal, or follow a linked file · close the modal
Copy-code button	`Enter` / `Space`	Copy the snippet
Zoomable image	`Enter` / `Space` · `Esc`	Open the lightbox · close it
Glossary term (dotted underline)	`Tab` · `Esc`	Focus the term to show its definition card · close the card
Twoslash token (typed code)	`Tab`	Focus a token in a Twoslash code block to reveal its inferred type; `Tab` away to hide
"Was this page helpful?" stars	← / →	Change the rating
Reader survey card (if enabled)	`Tab` · arrows · `Enter` / `Space`	Tab between the answers and **Submit**; arrows move within a choice group or the rating; `Enter` / `Space` selects or submits
"Ask AI" — open the panel (if enabled)	`Enter` / `Space` · `Cmd/Ctrl` + `I` · `Esc`	Open the chat from the header trigger, the bottom ask bar, or a page's **Ask a question** action; `Cmd/Ctrl` + `I` toggles it · `Esc` closes
"Ask AI" — bottom ask bar	type, then `Enter`	Ask from the bar pinned to the bottom of every page; it opens the panel and sends (shown only while the panel is closed)
"Ask AI" — from search results	↑ to the top row · `Enter`	The **Ask AI** option above the search matches opens the panel and asks your typed query
"Ask AI" — new chat	`Enter` / `Space`	The refresh button confirms before clearing the conversation

“Ask AI” answer feedback	`Enter` / `Space`	Rate an answer with the thumbs-up / thumbs-down buttons
Language selector · theme toggle	the control’s native keys	Switch language · light/dark

> **Search shortcuts:** the default theme doesn’t add a global search hotkey. If you enable Algolia DocSearch, it brings its own — `Cmd`/`Ctrl` + `K` and `/` — to open search from anywhere.

> **Ask AI assistant:** when the AI assistant is enabled, the chat panel is a non-modal dialog (`role="dialog"`, `aria-modal="false"`, an accessible name) — it deliberately does **not** trap focus, so the rest of the page stays operable behind it. `Esc` closes it; the thumbs-up/down feedback controls are labelled buttons. You can open it from the header trigger, the `Cmd`/`Ctrl` + `I` shortcut, a labelled **ask bar** fixed to the bottom of every page (a standard text field with a visible focus ring, shown only while the panel is closed so the two never overlap), or a page’s **Ask a question** action — and the **Ask AI** row at the top of the search results opens the panel and asks your typed query. The conversation persists per browser tab as you move between pages, and the panel’s **New chat** button asks to confirm before clearing it.

> **Reader survey:** when a survey is enabled, the prompt is an inline card at the end of the page — not a modal, so it never traps focus. Every control is a standard, labelled Mantine input (radios, checkboxes, a star rating, or a text box) plus a **Submit** button. As a multi-question survey advances, the new question is announced via an `aria-live="polite"` region, and the completion thank-you is announced the same way.

> **API reference:** the operation list lives in the documentation sidebar as standard links (`Tab` to reach, `Enter` to jump); the in-view operation is highlighted there as you scroll, and clicking one smooth-scrolls to it with a sticky-header offset — an instant jump when reduced motion is requested. Each request-sample **language** and response-example **status** picker is a labelled, keyboard-native Mantine select.

Build-time contrast check

The build reads your theme’s palette from `theme.scss`, checks it against WCAG AA, and prints a non-fatal warning for any text/background pair that falls short — in both light and dark:

```
Aardvark: warning — color contrast below WCAG AA:
  light — links: #8a7fff on #ffffff = 3.41:1 (needs 4.5:1)
  Adjust the colors in your theme's theme.scss so each pair meets the ratio
  (a darker/lighter shade per scheme), or set a11y.contrast: false to silence this.
```

It checks body text, secondary text, links, the active navigation item, text on the code/table surface, and the table header. Because each scheme has its own background, pick a `$primary` (and the other variables) that reads well on **both** — that’s why this site uses a lighter primary in dark mode than in light. Tune the check in `aardvark.config.yaml`:

```yaml
a11y:
  contrast: true        # run the audit (default: true; non-fatal warnings only)
  contrastLevel: AA     # AA (default) or AAA
# a11y: false           # shorthand to turn the audit off entirely
```

The audit runs against your theme's actual palette every build. The shipped default palette is already AA-clean, so a stock site stays quiet; a custom `theme.scss` that dips below the ratio gets a warning.

Writing accessible content

The theme handles the chrome; a few habits keep your pages accessible too:

- **Describe images** — always give Markdown images alt text: `![A labelled architecture diagram](/img.png)`. Use empty alt (`![]`) only for purely decorative images.
- **Keep headings in order** — one `#` per page, then `##`, then `###`; don't skip levels for size. The on-this-page list and screen-reader outline are built from them.
- **Write meaningful link text** — "read the deployment guide", not "click here".
- **Don't rely on color alone** — pair it with text or an icon when it carries meaning.
- **Label custom widgets** — any interactive snippet you add should be keyboard-operable and carry an accessible name (an `aria-label` or visible text) and a visible focus style.

> **Note:** Mantine components used in content (buttons, tabs, accordions, the rating widget) bring their own keyboard support and ARIA roles. Default filled buttons use Mantine's brand color; if you set a custom `$primary` in `theme.scss`, the contrast check above covers its use as link and active-state text.

Roadmap

- Live assistive-technology smoke tests (NVDA / VoiceOver) as part of releases.
- An automated `axe`/`pa11y` pass over the built sample site.

Versioned documentation

Aardvark can publish **multiple versions of your docs at once** — the latest release at the site root and older releases under a `/vX/` prefix — with a header version switcher, version-scoped search, and SEO that keeps the latest version as the canonical surface.

Versioning is **opt-in by subtree**: you choose which sections are versioned (say `/guide` and `/api`), and everything else — your landing page, blog, changelog — stays **global**, a single shared copy across versions.

How it works

Each older version is a **frozen snapshot committed to your repo** under `versions/<id>/`, and the whole site builds in one pass (no separate per-version builds). The current docs stay in `content/` and are served, unprefixed, at the root. It's the same model Docusaurus uses — and it reuses the exact machinery that powers multi-language sites, so a version is just a second URL prefix alongside the language one.

Enabling it

Declare which subtrees are versioned in `aardvark.config.yaml`. Nothing is versioned until you cut your first version, so this block is dormant on its own:

```yaml
versions:
  paths: [/guide, /api]        # only these subtrees are versioned; the rest stay global
  current:
    id: latest
    label: "2.0 (latest)"      # the working copy in content/, served at the root
  released:                    # managed by `vark version cut` (newest first)
    - { id: v1, label: "1.x" }
```

Cutting a version

When you're ready to freeze the current docs as a release, run:

```
vark version cut v1 --label "1.x"
```

This copies each `paths` subtree — across **every language** — into `versions/v1/<lang>/…` and adds the `released:` entry above. Because Aardvark builds each page's sidebar from its front matter, the snapshot's front matter is its frozen navigation. To see what's configured or to remove a version:

```
vark version list
vark version remove v1        # deletes the snapshot dir + config entry
```

Back-port a fix to an old version by editing its files directly under `versions/v1/`.

URLs

The default version is served unprefixed; older versions get a `/<id>/` prefix. On a multi-language site the language prefix stays outermost:

Version	Language	URL
latest	base	`/guide/intro/`
v1	base	`/v1/guide/intro/`
latest	French	`/fr/guide/intro/`
v1	French	`/fr/v1/guide/intro/`

A **version switcher** appears in the header next to the language picker. Switching keeps you on the same page in the target version when it exists there, or lands on that version's home when it doesn't. Older versions also show a persistent "you're viewing an older version" banner linking to the latest equivalent.

SEO and search

The latest version is the canonical surface. Older versions are set to `noindex`, their `<link rel="canonical">` points at the **latest equivalent**, and they're dropped from `sitemap.xml` and `llms.txt` — so search engines consolidate ranking on the current docs instead of treating each version as duplicate content. Opt an older version back into the index with `indexed: true` on its `released:` entry.

The on-page search box is **scoped to the version you're reading** (a reader on `/v1/` searches within v1), while crawler- and assistant-facing surfaces stay latest-only.

By default the version served at the root is `current` (your working `content/`). Pin a released version there instead — e.g. keep writing `3.0` under `/latest/` while `2.x` ships at the root — with `default: v2`.

Password protection

Aardvark builds a fully static site, so there's no server to check a login. The `protected` option instead **encrypts** every page under a directory at build time and ships an opaque blob plus a small loader page. Readers unlock it in the browser with a password; the plaintext is never published.

It's real encryption — **AES-256-GCM**, with the key derived from your password via **PBKDF2** (SHA-256, 600,000 iterations). The only thing standing between a visitor and the content is the password, so choose a strong one.

Try it live: this site has a demo at **/secretpage/** — open it and unlock it with the password `secret!` (shared on purpose, since it's just a demo). View source first if you like: you'll find only an encrypted blob until you enter the password.

Configuration

Add a `protected` list to `aardvark.config.yaml`. Each entry names a content-relative directory and the **environment variable** that holds its password at build time — you never put the password itself in the config:

```yaml
protected:
  - path: internal            # encrypts content/internal/**.md
    passwordEnv: DOCS_INTERNAL_PW
  - path: partners/secret     # a second area with its own password
    passwordEnv: PARTNERS_PW
```

You can protect any number of directories, each with a different password. The directory is matched on whole path segments, so `internal` protects `internal/` but not `internal-notes/`. If you nest entries (`internal` and `internal/secret`), pages use the **most specific** match's password.

Building

Set the password in the environment (a local, git-ignored `.env` works — `vark build` and `vark dev` load it automatically) and build as usual:

```
export DOCS_INTERNAL_PW='a strong passphrase'
vark build
```

If a protected directory's environment variable is **unset or empty**, the build **fails** rather than risk publishing the pages unencrypted — nothing is written and your previous build stays in place.

What readers see

Visiting a protected page shows a password box. On the correct password the real page is decrypted and rendered in place — fully interactive, exactly as if it had loaded normally. The password is remembered for the rest of the browser-tab session, so other protected pages in the same area open without re-prompting. After three wrong guesses the box stops asking; because the attempt count is saved in your browser, reloading the page won't reset it — to try again, clear the site's saved data in your browser. Where the browser won't let the page save that count — a sandboxed frame, a cookies-blocked profile, a full storage quota — the three-guess limit is still enforced, from memory, and the message tells you to reload the page to try again rather than to clear data that was never written.

On a multilingual site, a directory and all its translations share one password (protection is matched on the language-agnostic source path), so unlocking, say, `/en/internal/` in a tab also unlocks `/fr/internal/` — same content, same password.

Decryption uses the browser's WebCrypto API, which requires a **secure context**: the page must be served over **HTTPS** (or `localhost`). Over a `file://` path or plain HTTP on a LAN address the reader gets a "secure connection required" message instead of the prompt.

The decrypted page is rendered with `document.write`, so a site that sets a strict **Trusted Types** policy (`Content-Security-Policy: require-trusted-types-for 'script'`) will block the unlock — don't apply that CSP to protected pages.

A separate concern is a **nonce-based CSP** (`Content-Security-Policy: script-src 'nonce-…'`). A static-only deploy can't serve a real one — a nonce baked into a fixed `_headers` file is the same on every request, so it's guessable and non-functional. It works only behind a dynamic edge layer (Cloudflare Pages middleware/Worker, a Netlify Edge Function) that mints a fresh nonce per request and stamps it onto the loader's own script tag. In that setup Aardvark automatically propagates the loader's nonce onto the revived island module scripts, so they mount; without the propagation those scripts would be silently CSP-blocked. (A `'strict-dynamic'` policy needs no nonce on the island scripts — they're created by the already-trusted loader and inherit its trust.)

The loader also carries a tiny **inline** `<head>` script and stylesheet that apply your saved light/dark choice before the page paints, so a protected page doesn't flash the wrong scheme while the unlock client downloads. The stylesheet handles anyone following their system setting; the script is only needed to honor an explicit choice that differs from it. Each is covered by its own CSP directive, so a nonce policy can cost you one or both: if your edge layer stamps its nonce only onto script tags with a `src`, `script-src` blocks the inline script and just the readers who overrode their system setting see the old flash — but a nonce-based **`style-src`** blocks the inline stylesheet as well, and then everyone does. Stamp the request nonce onto both inline elements (or allow them explicitly) to keep the pre-paint fix.

What stays private — and what doesn't

Protected pages are kept out of every discovery surface Aardvark emits: the **sitemap**, **on-site search index**, `llms.txt` / `llms-full.txt`, **hover cards**, **Open Graph cards**, **RSS feeds**, and their plaintext `.md` **siblings**. Search engines and the AI assistant never see their contents.

Two things are not encrypted, so don't rely on them being secret:

- **The URL.** The loader page lives at the page's normal address, so the path itself is discoverable. The content is encrypted; the location isn't.
- **Navigation labels.** If a protected page appears in your sidebar or is linked from a public page, its title/link text is ordinary chrome and remains visible. Keep secrets in the page body, not in titles or filenames.

Security notes

- **The password is the whole defense — make it strong.** The page key is derived with PBKDF2 (SHA-256, 600,000 iterations — the OWASP 2023 minimum). Anyone can download the `.enc` blob and try passwords offline on fast hardware, so a short or dictionary password can be cracked. Use a long, random passphrase, and treat this as protection for low-to-moderate-sensitivity content — not a substitute for server-side authentication on truly sensitive data.
- **The unlock is cached in the browser for the tab session.** After a successful unlock the derived key — not your password — is stored in `sessionStorage` (and cleared when the tab closes), so other protected pages in the same area open instantly without re-prompting. Like anything in web storage it is readable by JavaScript on your own origin, but it is only the key for that one area (it can't recover your password or unlock anything re-encrypted later). Even so, don't combine protected pages with untrusted third-party scripts.
- **At build time the password is a normal build secret.** It's read from its env var and lives in the build process's memory (and environment) for that run — readers never receive it (only the derived key and ciphertext ship). Treat the env var like any deploy secret: keep it out of shell history and CI logs, and prefer your platform's secret store over a checked-in value.
- **Encrypted blobs are served `no-store`.** Each rebuild mints a fresh salt, so the `.enc` payload changes — but the cached key (above) would still decrypt a stale blob carrying the old salt, silently showing outdated content in an open tab. To prevent that, Aardvark adds a `Cache-Control: no-store` rule for `/*.enc` to the generated `_headers` file, so the browser and CDN always fetch the current blob (a rotated salt then re-prompts instead). The `_headers` file is honored by **Cloudflare Pages** and **Netlify**, whose `*` wildcard also matches across `/` so the rule reaches nested directories. Other targets — **Vercel** (`vercel.json` headers), a plain **nginx**/Apache config, or any **self-host** — don't read `_headers`, so add an equivalent `Cache-Control: no-store` on `.enc` files there yourself. A few hosts — notably **GitHub Pages** — expose no custom-header mechanism at all, so you can't set `no-store` there; that's still safe (the blobs are encrypted), but after you rotate a password an already-unlocked reader keeps seeing the old content until their cached copy expires or they close the tab. Skipping the header is never a confidentiality risk — only a staleness one.

Deploy Aardvark

`vark build` writes a plain static site to `build/` — no server required. Host that directory on any static host, or run it yourself with the bundled `vark serve`. This section collects everything about shipping Aardvark to production.

New here?

Install the CLI first — see Installation — then come back to ship what you've built.

Deployment

Hand the static `build/` directory to any host — Cloudflare Pages, Netlify, Vercel, GitHub Pages, S3 — or ship Aardvark itself as one compiled binary.

CLI reference

`build`, `dev`, `serve`, `link-check`, and the rest — and the exact flags each command takes.

Generated files

`sitemap.xml`, `robots.txt`, `_redirects`, `_headers`, per-page Markdown, the search index, and the whole-site PDF.

Run a live server

Need an origin process — for the MCP endpoint, `Accept: text/markdown` negotiation, or a CDN to sit in front of? That's `vark serve`. It lives under **AI Features** because the same process also exposes your docs to agents over MCP.

Self-hosting & MCP

The hardened `vark serve` process — static plus a live MCP server — packaged as a Docker image behind a CDN. Documented under AI Features.

Deployment

Hosting the site

`vark build` produces a fully static `build/` directory — HTML, one JS/CSS island bundle, your assets, and the generated files. Deploy it to any static host:

- **Netlify / Vercel / Cloudflare Pages** — build command `vark build`, publish directory `build`. Node must be available in the build image (it is on all three) to bundle islands.
- **GitHub Pages / S3 / nginx** — build in CI, then upload `build/`.
- **Run it yourself** — `vark serve` serves the build (and an optional live MCP server) from one hardened process, ready to package as a Docker image behind a CDN.

Set `baseUrl` in `aardvark.config.yaml` to your production URL so `sitemap.xml` and `llms.txt` use absolute links.

Caching

Cache fingerprinted assets for a long time. Root CSS/JS, files copied from `static/` or `public/`, theme/runtime assets under `/_aardvark/`, islands bundles, generated PWA icons/manifests, OG images, and controlled downloads are emitted with per-build filenames such as `/img/logo-<sha>.png`.

Keep route and discovery files revalidatable instead: page HTML, `_headers`, `_redirects`, `robots.txt`, `sitemap.xml`, `.well-known/**`, `auth.md`, `llms*.txt`, `metadata.json`, `search-index.json`, page `.md` siblings, and encrypted `.enc` payloads stay at stable paths by design.

The fingerprint token is your git `HEAD` SHA. If you build where `.git` isn't present (a container image, or a CI job with a shallow/absent checkout), set the `AARDVARK_BUILD_SHA` environment variable to the commit being deployed so every build still gets a stable, unique token. Without it, Aardvark falls back to a content hash of your project sources — correct, but it rotates every asset URL whenever any source file changes.

Redirects

Page `aliases:` and config `redirects:` (see Generated files) make `vark build` write a `_redirects` file. **Cloudflare Pages, Netlify, and Vercel** read it and serve true `301`s; other hosts ignore it, but per-page aliases still work everywhere through their HTML stub pages. On Cloudflare/Netlify a static file takes precedence over a `_redirects` rule for the same path (Netlify can force it with a trailing `!`, Cloudflare can't), so an alias's stub wins there unless you disable stubs with `aliases: { htmlStubs: false }`.

Publishing under a subpath

By default a site owns the root of its domain (`https://docs.example.com/`). To publish it as a section of a larger site instead — say at `https://example.com/docs/` — set `basePath`:

```
basePath: /docs
```

Every internal link, asset URL, redirect, localized URL, the search index, and the on-disk layout are then written for that prefix: pages land under `build/docs/…`, links point at `/docs/…`, and the whole `build/docs` folder is self-contained — drop it into a parent site at `/docs`, or serve it with `vark serve` / `vark dev` (both mount at the base path locally, so local dev matches production). If `baseUrl` is also set it keeps carrying the origin (`https://example.com`); Aardvark folds the base path into the absolute URLs used by `sitemap.xml`, canonical tags, and Open Graph so they stay correct.

Authors keep writing site-relative links (`/guide/intro/`) — the base path is added at build time, so content stays portable if the prefix ever changes.

> **Password-protected pages:** each encrypted page is cryptographically bound to its own URL, which includes the base path. Adding or changing `basePath` on a site that already has protected pages therefore requires a fresh full build (which re-encrypts every protected page) — a stale `.enc` blob from before the change is bound to the old path and will fail to decrypt. A normal `vark build` + redeploy handles this; just don't ship an old `.enc` alongside the new loader page.

Where the origin-root files go

`robots.txt`, `_redirects`, `_headers`, and everything under `/.well-known/` are only honored at the true origin root. `rootFiles` controls where they land:

```
basePath: /docs
rootFiles: nested   # nested (default) | root
```

- **`nested`** (default) — self-contained: these files nest under `build/docs/` with everything else. Use when you drop the folder into a parent site that owns the domain's real `robots.txt` / `/.well-known/`.
- **`root`** — writes those files at the true build root (with their paths pointed into `/docs/…`) while the pages nest under `build/docs/`. Use when this build is deployed as its own project that owns the origin but serves the app under `/docs`.

Custom 404 page

Create `404.md` at the root of your `content/` directory and Aardvark renders it to `404.html` at the site root — wrapped in your theme, with the header, footer and search box, just like any other page:

```
---
title: Page not found
description: We couldn't find that page.
---

# Page not found

The page you're looking for doesn't exist. Head back to the [home page](/).
```

It's authored like a normal page (Markdown, `` tags and components all work), but it's treated specially: it's served at the literal `404.html` instead of a pretty `/404/` URL, marked `noindex`, and kept out of the nav, sitemap, search index and `llms.txt`. Only a `404.md` at the content **root** is special — a nested `guide/404.md` stays an ordinary `/guide/404/` page.

Most static hosts serve `/404.html` automatically for unmatched URLs (GitHub Pages, Netlify and Cloudflare Pages do out of the box; on S3 set the error document to `404.html`; on Cloudflare Workers static assets set `assets.not_found_handling` to `"404-page"`). `vark dev` serves it too, so you can preview your error page by visiting any URL that doesn't exist.

Multilingual sites: put your `404.md` at the **base-language** content root — that's the `/404.html` every host serves for unmatched URLs, so it's what visitors see regardless of the URL they tried. A `404.md` in a translated source dir (e.g. `content-fr/404.md`) is still built, to `/fr/404.html`, but most hosts ignore it and fall back to the root page. The exception is Cloudflare Workers' `"404-page"` mode, which serves the nearest `404.html` up the path — so there a miss under `/fr/` returns `/fr/404.html` when it exists. Unless your host does that nearest-match lookup, a single base-language `404.md` is all you need.

Shipping Aardvark as a binary

Compile the tool itself to a single executable with Nuitka:

```
scripts/build-release.sh    # -> dist/vark (single file)
```

The output lands in `dist/` by default; set `BUILD_OUT_DIR` to write it elsewhere.

(For local iteration, `scripts/build-binary.sh` is a faster non-`--onefile` build that outputs a folder instead of a single file.)

The binary bundles the default theme and the islands runtime, so it works with no Python environment. **Node remains a runtime prerequisite** on whatever machine runs `vark build`, because islands are bundled with esbuild at build time. Use `vark build --no-bundle` if you need to build without Node (components become inert placeholders).

CLI reference

Aardvark — a Mantine-powered static site generator.

Author your content in Markdown and build it into a fast, fully static site: Mantine-powered interactive components, built-in client-side search, automatic Open Graph cards, and optional build-time AI enrichment. Scaffold a project with `vark new`, preview it live with `vark dev`, and produce the deployable site with `vark build`.

On interactive terminals, vark quietly checks for newer releases in the background while real subcommands run, and prints an upgrade reminder only when an update is available. Run `vark update` any time for an explicit check.

Run with no command to open the interactive TUI (agentic authoring + launchers).

Run `vark <command>` (or `aardvark <command>`) from your project root.

Examples:

```
vark new my-docs        # scaffold a new site in ./my-docs
cd my-docs && npm install   # install the islands toolchain (needs Node)
vark dev                # preview at http://localhost:8000, live-reload
vark build              # build the static site into ./build
```

Commands

Command	Description
`vark new`	Scaffold a new site in PATH.
`vark build`	Build the site to the output directory.
`vark dev`	Serve the site locally and rebuild on change.
`vark link-check`	Check internal links, anchors, and images across the site (a build smoke test).
`vark convert`	Import a docs site built for another platform into an Aardvark site.
`vark ai-enrich`	Run opt-in build-time enrichment (frontmatter, examples, skills).
`vark author`	Run agentic authoring tasks on your docs — interactively or headless.
`vark serve`	Serve a built site (and a live MCP server) for production.
`vark update`	Check whether a newer version of Aardvark is available.
`vark version`	Manage documentation versions (snapshot-in-tree docs versioning).
`vark web-bot-auth-keygen`	Generate an Ed25519 keypair for Web Bot Auth.

`vark new PATH`

Scaffold a new site in PATH.

Creates PATH and fills it with starter content, a sample data file, the editable default theme under `themes/vark/`, an example snippet, and a `package.json` for the Mantine islands toolchain. Next: `cd` into PATH and run `vark dev` — when Node is available the first build sets up the islands toolchain with npm (or run `npm install` yourself up front); without Node it builds without the islands bundle.

Examples:

```
vark new my-docs   # scaffold ./my-docs, then `cd my-docs`
```

`vark build`

Build the site to the output directory.

Every build prints a per-phase timing summary — how many pages and components were produced, and how long each phase (discovery, render, island bundling, …) took. The summary goes to stdout and the live `--verbose` progress to stderr, so `vark build > summary.txt` keeps the digest while progress still streams to the terminal. Conditional phases (translation, AI, OpenAPI, island bundling) appear only when they actually run.

A link to a missing page or a missing local image — in any language — always fails the build (the same check `vark link-check` runs without producing output). A link to a missing `#anchor` on a page that DOES exist is warn-only by default (it still lands on the right page); set `links: {strictAnchors: true}` in config to make those fail too. Unknown component references never fail the build: each renders as an HTML comment and is reported as a warning on stderr — usually a typo, a missing `npm install`, or a snippet you haven't created yet.

`--log-file PATH` also appends this build's summary, warnings, and any failure to a plain-text file — what `vark build > summary.txt` captures, plus the warnings and the error (`-v`'s per-phase streaming stays on stderr and is not written to the file). It exists mainly for symmetry with `vark dev --log-file`, where a pipe isn't an option because it turns the dashboard off. Runs are appended under a `=== vark build … started … ===` banner.

Option	Default	Effect
`--root DIRECTORY`	`.`	Project directory (defaults to the current directory).
`--bundle / --no-bundle`	`--bundle`	Build the islands JS bundle.
`--pdf / --no-pdf`	`--pdf`	Render the whole-site PDF when enabled in config (off in `vark dev`).
`--pdf-reuse / --no-pdf-reuse`	`--pdf-reuse`	Reuse the live PDF within `pdf.reuseForDays` instead of re-rendering. --no-pdf-reuse forces a fresh render.
`--model TEXT`	—	Override the model for AI features (a gateway model slug).
`--translate / --no-translate`	`--no-translate`	Refresh missing + changed translations with Claude (requires AARDVARK_SECRET_KEY).
`--retranslate-all`	off	Re-translate every page, overwriting existing translations (implies --translate).

`--generators / --no-generators`	`--generators`	Run generation scripts in generators/ (write Markdown pages before the build).
`-v, --verbose`	off	Stream per-phase progress as the build runs.
`--plain / --no-plain`	auto	Force plain (or rich) output. Auto-detected off a TTY, or under CI / NO_COLOR.
`--log-file PATH`	—	Append this build's log — the summary, warnings, and any failure — to PATH as plain text. Off unless you pass it.

Examples:

```
vark build               # build ./ into ./build
vark build --no-bundle   # skip the islands JS bundle (no Node needed)
vark build -v            # stream per-phase progress to stderr
```

`vark dev`

Serve the site locally and rebuild on change.

Builds once, serves the result, watches your sources, and live-reloads the browser on every change. Your browser opens to the site automatically when the server starts (pass `--no-open` to skip that). The per-phase timing summary prints on the first build and on every rebuild, so you can see exactly what each change cost. Generation scripts run on every rebuild; pass `--no-generators` for a fully offline loop when a generator's network call is slow or its cache is cold.

When the site prerenders islands (`islands.ssr`), each page's baked markup is cached and reused until that page — or the islands toolchain — actually changes, so a rebuild only re-renders what you edited. Pass `--no-ssr` to skip the prerender altogether for the fastest loop, accepting that dev pages then mount client-side and no longer match what `vark build` publishes.

The live dashboard paints on the terminal's alternate screen, so its log vanishes when you quit and can't be piped anywhere (piping `vark dev` turns the dashboard off entirely). Drag-select copies what the pane still shows (2,000 wrapped lines); `s` saves from a separate, larger record of 5,000 unwrapped lines, so it keeps output the pane has already scrolled past. Both are still capped. Pass `--log-file dev.log` to also write the same events — every build summary, warning, broken-link report, and traceback — to a plain-text file you can paste from, with no cap at all. Runs are appended, each under a `=== vark dev … started … ===` banner, so restarting never discards the output you were trying to keep.

Option	Default	Effect

`--root DIRECTORY`	`.`	Project directory (defaults to the current directory).
`--port INTEGER`	`8000`	Port to serve the site on.
`--open / --no-open`	`--open`	Open the site in your browser when the server starts.
`--generators / --no-generators`	`--generators`	Run generation scripts on each rebuild. Pass --no-generators for a faster, fully offline dev loop when a generator makes a slow/uncached network call.
`--ssr / --no-ssr`	auto	Prerender islands on each rebuild (defaults to the site's islands.ssr setting). Pass --no-ssr to skip both the SSR renderer build and the prerender: islands then mount client-side, so dev pages flash on load and no longer match production output.
`-v, --verbose`	off	Stream per-phase progress on every rebuild.
`--plain`	off	Disable the live dashboard; use plain line-by-line output (also implied off a TTY or under CI / NO_COLOR).
`--log-file PATH`	—	Append this run's log — builds, warnings, and errors — to PATH as plain text, so the dashboard's output can be copied into a bug report. Off unless you pass it; PATH must be outside the build output directory, which every build replaces.

Examples:

```
vark dev               # serve ./ at http://localhost:8000, live-reload
vark dev --port 3000   # serve on a different port
vark dev --no-open     # don't open a browser when the server starts
vark dev --log-file dev.log   # also append the log to a file you can paste from
```

`vark link-check`

Check internal links, anchors, and images across the site (a build smoke test).

Renders every page in memory and verifies that internal links, `#anchors`, and local images resolve, without writing HTML output — the same check `vark build` runs. Use it for a fast pass/fail in CI or a pre-commit hook. A link to a missing page or a missing local image is printed to stderr and fails (exit non-zero). A link to a missing `#anchor` on a page that DOES exist is a warning by default (still exits 0) — set `links: {strictAnchors: true}` in config to make those fail too. Generation scripts

run first so generated pages are checked too; pass `--no-generators` to skip them (e.g. an offline check of a site whose generators call the network).

Image checking covers Markdown images, raw `<img>` / `srcset` / `<video poster>` references, CSS background images, and built-in image-bearing islands. It validates only local files the site would publish from `static/`, `public/`, and theme assets; external and data-URI images are skipped.

Option	Default	Effect
`--root DIRECTORY`	`.`	Project directory (defaults to the current directory).
`--generators / --no-generators`	`--generators`	Run generation scripts before checking (so generated pages are link-checked too). --no-generators is a fully read-only check that writes nothing to content/ — use it in CI when generated pages are already committed/cached, or for an offline check.
`--plain / --no-plain`	auto	Force plain (or rich) output. Auto-detected off a TTY, or under CI / NO_COLOR.

Examples:

```
vark link-check   # verify internal links and images without writing output
```

`vark convert PLATFORM INPUT_PATH OUTPUT_PATH`

Import a docs site built for another platform into an Aardvark site.

Reads the source docs at INPUT_PATH (a project built for PLATFORM), translates its navigation, pages, components, and assets into Aardvark's format, and writes a buildable site to OUTPUT_PATH — an `aardvark.config.yaml`, `content/*.md`, and `static/` you can immediately `vark build` or `vark dev`. Conversion is best-effort: the summary reports how many pages and assets were converted and flags any construct that didn't map cleanly, so you know exactly what to review by hand.

The supported PLATFORM values are named in the epilog below (and running with an unknown one lists them). OUTPUT_PATH must be empty unless `--force` is given.

Argument	Possible values
`PLATFORM`	`docusaurus`, `fern`, `gitbook`, `hugo`, `mintlify`, `mkdocs`, `readme`
`INPUT_PATH`	—

`OUTPUT_PATH`	—

Option	Default	Effect
`--force`	off	Write into OUTPUT_PATH even if it already exists and is non-empty. The top-level entries this conversion generates (aardvark.config.yaml, content/, static/, and any converter dirs such as openapi/) are replaced wholesale — files left inside content/ or static/ by a previous run do NOT survive, so stale pages/assets can't linger. Other top-level entries the conversion doesn't produce are left untouched. Use an empty directory if you want output to match the source exactly.
`-v, --verbose`	off	List each unconverted construct in the summary, not just the count.

Supported platforms: docusaurus, fern, gitbook, hugo, mintlify, mkdocs, readme.

Examples:

```
vark convert mkdocs ./my-mkdocs-site ./my-docs   # import a MkDocs site
vark convert mkdocs ./mkdocs.yml ./my-docs        # point at the config directly
vark convert gitbook ./my-gitbook-repo ./my-docs # import a GitBook repo
vark convert mkdocs ./src ./out --force -v        # overwrite a non-empty ./out
```

`vark ai-enrich`

Run opt-in build-time enrichment (frontmatter, examples, skills).

Writes generated `description`/`keywords` back into your Markdown frontmatter and, when `ai.skills` is enabled, (re)generates a `SKILL.md` for each planned skill under `skills/` in your project root. This is the only command that touches skills — `vark build` leaves them alone. Runs through Aardvark's metered gateway: the only thing you need to set is your `AARDVARK_SECRET_KEY`.

Option	Default	Effect
`--root DIRECTORY`	`.`	Project directory (defaults to the current directory).
`--model TEXT`	—	Override the model for AI features (a gateway model slug).
`--plain / --no-plain`	auto	Force plain (or rich) output. Auto-detected off a TTY, or under CI / NO_COLOR.

Examples:

```
vark ai-enrich   # enrich frontmatter and (re)generate planned skills
```

`vark author`

Run agentic authoring tasks on your docs — interactively or headless.

With no `--action`, opens a menu of AI-assisted actions — regenerate keywords, refresh a page's description, check and replace dead outbound links, apply a style-guide pass, or chat with a writing agent — that edit your Markdown in place (each change previewed as a diff you confirm before it's written). The same menu is reachable by running bare `vark`.

With `--action <id>` it runs ONE action non-interactively (for CI / automation), previewing diffs unless `--yes` is given to write. Scope it with `--path <file-or-dir>` (repeatable), `--diff <ref>` (only pages changed vs a git ref), or `--page` for one; with none of those it runs on every page. This is the mode the gateway's GitHub integration drives on a runner. The styleguide action additionally REQUIRES `--rulesets` — the style ruleset ids to apply, comma-separated in precedence order (list order, first wins conflicts); no other action accepts the flag.

Running an action (or the interactive menu) requires your Aardvark **secret** key in `AARDVARK_SECRET_KEY` — the possession-based CLI credential; every call goes through Aardvark's metered gateway, never a model provider directly. (`--list-actions` only prints the headless action ids and needs no key.)

Option	Default	Effect
`--root DIRECTORY`	`.`	Project directory (defaults to the current directory).
`--model TEXT`	—	Override the model for the agent (a gateway model slug).
`--page TEXT`	—	Pre-select a page by content-relative path (e.g. guide/intro.md) or URL, skipping the picker.
`--action TEXT`	—	Run ONE authoring action non-interactively across pages (headless / CI). See --list-actions for the available ids.
`--all`	off	With --action, run on every page (the default when no --path/–diff is given).
`--path TEXT`	—	With --action, scope to a file or directory (repeatable). A file targets that page; a directory targets every page under it. Omit --path/–diff/–all to run on every page.

`--diff TEXT`	—	With --action, scope to only the pages whose source changed vs a git ref (e.g. --diff main = pages this branch changed). Runs nothing if no page changed.
`-y, --yes`	off	With --action, WRITE the changes to disk. Without it, diffs are previewed only.
`--rulesets TEXT`	—	With --action styleguide: comma-separated style ruleset ids to apply, highest precedence first (microsoft, google, alex, writegood, proselint, readability).
`--list-actions`	off	List the authoring actions that can run headless via --action, then exit.
`--plain / --no-plain`	auto	Force plain (or rich) output. Auto-detected off a TTY, or under CI / NO_COLOR.

Examples:

```
vark author                                # open the agentic authoring menu
vark author --page guide/intro.md          # act on one page directly
vark author --list-actions                 # list actions runnable headless
vark author --action styleguide --rulesets microsoft,google
                                           # preview a style pass on every page
vark author --action keywords --path guide/ --yes
                                           # apply to just the guide/ subtree (CI)
vark author --action keywords --diff main --yes
                                           # apply to only the pages this branch changed
```

`vark serve`

Serve a built site (and a live MCP server) for production.

The hardened, headless counterpart to `vark dev`: a single uvicorn/Starlette process that serves the already-built `./build` — honoring the build's own `_headers` / `_redirects` so content types, redirects, and the `ai-config.json` no-store rule match a CDN host — and, unless `--no-mcp`, exposes a stateless MCP server at `/mcp` that wraps the site's docs as read-only tools (search, fetch, list, full corpus).

It does NOT build, watch, or live-reload — run `vark build` first. Designed to sit behind a CDN, which terminates TLS and absorbs the static volume; the per-IP `/mcp` rate limit is a single-process backstop, not a DDoS shield. For full-text MCP search on a site without the on-page search box, set `mcp: true` in `aardvark.config.yaml` before building.

The serve stack (starlette/uvicorn/mcp) ships with every Aardvark install — pip, the standalone binary, and the Docker image — so there is nothing extra to install.

Option	Default	Effect
`--root DIRECTORY`	`.`	Project directory (defaults to the current directory).
`--host TEXT`	`0.0.0.0`	Interface to bind (default: all interfaces).
`--port INTEGER`	`8080`	Port to serve on.
`--workers INTEGER`	`1`	uvicorn worker processes (default 1).
`--mcp / --no-mcp`	`--mcp`	Expose the live MCP server at /mcp.
`--trusted-proxy CIDR`	—	CIDR(s) of a CDN/load balancer to trust for X-Forwarded-For / CF-Connecting-IP (repeatable). Default: loopback only — so behind a CDN, set this or all clients share one rate-limit bucket. Pass `--trusted-proxy none` to trust no proxy at all (not even loopback; forwarded headers are then never honored).
`--mcp-rate-limit INTEGER`	`60`	Per-IP requests/minute for /mcp (default 60), PER WORKER — with --workers N the effective per-IP cap is N x this (no shared store across workers). An invalid value falls back to 60.

Examples:

```
vark serve                       # serve ./build on 0.0.0.0:8080, with /mcp
vark serve --port 80 --no-mcp    # static only, no MCP endpoint
vark serve --trusted-proxy 173.245.48.0/20  # trust a CDN's forwarded IPs
```

`vark update`

Check whether a newer version of Aardvark is available.

Queries the public Homebrew tap for the latest published release. If you're behind, it prints how to upgrade — `brew upgrade aardvark`, or a direct binary download for your Mac (for people who don't use Homebrew). It only checks and reports; it never touches your installation, and degrades gracefully when offline.

Examples:

```
vark update   # check for a newer release and how to upgrade
```

`vark version COMMAND [ARGS]...`

Manage documentation versions (snapshot-in-tree docs versioning).

Declare which subtrees are versioned in your config first (`versions.paths`, e.g. `[/guide]`), then `vark version cut <id>` freezes a copy of those subtrees under `versions/<id>/` and serves it under `/<id>/`. The latest docs stay in `content/` and are served at the site root. See the versioning guide.

Examples:

```
vark version cut v1        # freeze the current docs as version v1
vark version list          # show configured versions
vark version remove v1     # delete the v1 snapshot + config entry
```

`vark web-bot-auth-keygen`

Generate an Ed25519 keypair for Web Bot Auth.

Prints the public key to add under `webBotAuth.keys` in `aardvark.config.yaml` (so `vark build` publishes it at `/.well-known/http-message-signatures-directory`) and the private key to store as a secret for whatever agent signs requests on your site's behalf. Aardvark only ever publishes the public directory — it never stores the private key, so save it now; it is not recoverable.

Option	Default	Effect
`--json`	off	Emit the keypair as JSON instead of text.

Examples:

```
vark web-bot-auth-keygen          # print a keypair + a paste-ready config snippet
vark web-bot-auth-keygen --json   # machine-readable {privateKey, x, kid}
```

Generated files

Every `vark build` writes these automatically:

`sitemap.xml`

A standard sitemap of every page, using `baseUrl` from `aardvark.config.yaml`.

`robots.txt`

Allows all crawlers, points them at the sitemap, and declares **Content Signals** — a machine-readable statement of how automated systems may use your content, per contentsignals.org and its IETF draft. A short comment block explains the policy, and one `Content-Signal:` line carries it:

```
User-agent: *
Content-Signal: search=yes, ai-input=yes, ai-train=yes
Allow: /
```

The three signals are:

- **`search`** — building a search index and showing links and short excerpts in results (not AI-generated summaries).
- **`ai-input`** — using the content in AI models in real time (retrieval-augmented generation, grounding, generative search answers).
- **`ai-train`** — training or fine-tuning AI models.

Each is `yes` or `no`. By default Aardvark grants all three. To change that, add a top-level `robots:` block — distinct from `seo.robots`, which is the per-page `<meta name="robots">` directive:

```yaml
robots:
  contentSignals:
    search: yes      # listed signals override; unlisted ones stay at the yes default
    ai-input: yes
    ai-train: no     # reserve training rights while still allowing search + AI answers
```

Listed signals override the default; an unlisted signal stays `yes`. Set one to `no` to deny that use, or to `null` to drop it from the line (declaring no preference — neither granting nor restricting it).

To omit Content Signals entirely, set `robots: false` (or `robots: { contentSignals: false }`) — only an explicit `false` (or `off`/`0`) disables. A bare or empty `contentSignals:` is read as "not configured" and falls back to the all-yes default, so a placeholder key never silently strips the signals. Either way the rest of the `robots.txt` is still written.

As with every artifact here, a hand-written `static/robots.txt` (or `public/robots.txt`) is copied verbatim and **always wins** — Aardvark won't overwrite it.

`llms.txt` and `llms-full.txt`

Per the llmstxt.org convention, to make your docs easy for LLMs to consume:

- **`llms.txt`** — an index: your site name, summary, and a linked list of every page with its description. Each entry links to the page's raw `.md` (below) so an assistant gets clean Markdown, not rendered HTML — or to the HTML page when per-page Markdown is turned off.
- **`llms-full.txt`** — the full content of every page concatenated, with island markup stripped to clean text.

`site.summary` (or `site.description`) in your config becomes the `llms.txt` summary; each page's `description` is used in the index list.

`search-index.json`

A full-text index — one record per page (URL, title, breadcrumb, description, keywords, headings, inbound link text, and body text) — that powers the built-in search. Always written unless you set `search: false`.

These need no configuration beyond `baseUrl` and your page frontmatter.

Web Bot Auth directory

When you enable `webBotAuth`, the build also publishes a key directory (a JWKS) at `/.well-known/http-message-signatures-directory`, so your site can identify itself when an agent sends signed requests on its behalf. It's off until you add a key — see Web Bot Auth.

`.well-known/agent-skills/index.json`

If your project has a `skills/` directory, every `skills/<name>/SKILL.md` is published as an **Agent Skills Discovery** index at `/.well-known/agent-skills/index.json`, per the Agent Skills Discovery RFC (v0.2.0). Agents that support the convention can then discover the skills your site offers — the way `llms.txt` and `sitemap.xml` advertise your content.

The index lists each skill with its name, `type`, description, URL, and a SHA-256 content digest:

```
{
  "$schema": "https://schemas.agentskills.io/discovery/0.2.0/schema.json",
  "skills": [
    {
      "name": "deploy-a-docs-site",
      "type": "skill-md",
      "description": "Build an Aardvark documentation site and deploy it to a static host.",
      "url": "/.well-known/agent-skills/deploy-a-docs-site/SKILL.md",
      "digest": "sha256:…"
    }
  ]
}
```

Each `SKILL.md` is copied verbatim to `/.well-known/agent-skills/<name>/SKILL.md` (the `<name>` is its source directory), and its `digest` is the SHA-256 of that file's bytes, so a client can verify what it fetched. The skills come from your `skills/` directory — generated by `vark ai-enrich` or hand-authored; either way `vark build` just publishes whatever is on disk, with no AI configuration required. The build also synthesizes a "how to use this site" agent-help skill, so the index is published on every build whenever the Markdown menu is on (the default) — even with no `skills/` directory. It is omitted only when there are no skills at all (no `skills/` directory and `markdownMenu` off).

As with every artifact here, a hand-written `static/.well-known/agent-skills/index.json` always **wins** — Aardvark then leaves the whole `.well-known/agent-skills/` tree untouched.

Agent discovery — `Link` headers

So an AI agent landing on your site can find the machine-readable files above, every build sets an RFC 8288 `Link` response header on the home page (/). It's written into the same `_headers` file, so it works on hosts that read it (Cloudflare Pages, Netlify) and under `vark serve`, and points at each resource with a registered relation type — all on one comma-separated header:

- `</llms.txt>` and `</llms-full.txt>` — `rel="service-desc"`
- `</sitemap.xml>` — `rel="describedby"`
- `</metadata.json>` — `rel="service-desc"`, when the AI assistant or MCP is on and `markdownMenu` is enabled (the build only writes `metadata.json` when per-page Markdown is published)
- `</.well-known/agent-skills/index.json>` — `rel="service-desc"`, whenever the agent-skills index is published (with the Markdown menu on, that's every build — above)
- `</.well-known/api-catalog.json>` — `rel="api-catalog"`, when the site has an OpenAPI reference (below)

No configuration. To override it, declare a `/` block in your own `static/_headers`. A site-wide `seo.noindex` (a staging build) suppresses it, matching how `noindex` keeps a build out of every other discovery surface.

`api-catalog` for OpenAPI sites

If any page embeds an `{% openapi %}` reference, the build also writes an RFC 9727 catalog at `/.well-known/api-catalog.json` — an `application/linkset+json` document listing each API with a machine-readable `service-desc` link to the spec (republished as JSON under `/_aardvark/openapi/`) and a `service-doc` link to its docs page. A request to the canonical extensionless `/.well-known/api-catalog` redirects there. Nothing to configure — it appears only when you actually have a spec.

Per-page Markdown and the “View in Markdown” button

Every page is also written as raw Markdown beside its HTML, so readers — and LLMs — can grab the source. A page at `/guide/intro/` is served at `/guide/intro.md`; the home page is `/index.md`. It’s the same processed Markdown that feeds `llms-full.txt`, with island markup stripped to clean text.

To surface it, each page shows a **View in Markdown** button at the top-right of the content. Clicking the label opens the page’s `.md`; the chevron beside it opens the rest:

- **Copy page** — copy the Markdown to the clipboard.
- **Download PDF** — download the whole site as a single PDF. Shown only when `pdf: true` is set (see below).

The dropdown also carries the **agent hand-off** items — Copy MCP Server, Install Skill, Install Plugin, and Install Assistant — covered on the Agent readiness page. Aardvark also writes a `_headers` rule so hosts that read it (Cloudflare, Netlify) serve the `.md` as `text/plain` — shown inline rather than the default `text/markdown`, which browsers download — and the generated `.txt` files (`llms.txt`, `llms-full.txt`, `robots.txt`) as `text/plain; charset=utf-8`, so characters like em-dashes render correctly instead of as mojibake. Hosts that ignore `_headers` (Vercel, GitHub Pages, …) still serve the files; add your own header rule there if you want inline display and UTF-8.

Tuning

On by default. A bare `markdownMenu: false` turns off **both** the `.md` files and the button; otherwise toggle individual actions or set the button label:

```yaml
markdownMenu:
  enabled: true               # gates the .md files AND the button
  viewMarkdown: true          # per-item toggles (all default true)
  copyMarkdown: true
  downloadPdf: true
  label: View in Markdown     # the primary button's label
```

The agent hand-off items have their own toggles (`copyMcp`, `installSkill`, `installPlugin`, `installAssistant`) — see Agent readiness.

PDF output

Set a top-level `pdf: true` to render the **whole site to one downloadable PDF** — a cover page, a clickable table of contents, then every page in nav reading order with a bookmark per page — named after your site (`Aardvark` → `/aardvark.pdf`). A **Download PDF** item is added to the dropdown on every page, linking to that single document. It's laid out as a clean, printable handbook with syntax-highlighted code (long lines wrap) and tables. **Built-in components are re-rendered for print** — an `{% openapi %}` reference becomes operation/parameter tables, a card or callout a titled block, and so on — so interactive widgets aren't blank on paper; a live map gets a "view online" note, and a component from a React library you import is left to its plain content. Links between your own pages become in-document jumps (so the PDF reads on its own), while links to other sites stay clickable. The cover shows your theme logo, site name, description, base URL and a build timestamp. Password-protected pages are never included (their content ships encrypted). It's off by default since rendering adds a little build time.

```
pdf: true
```

To keep generating the PDF but hide the menu link, set `markdownMenu: { downloadPdf: false }`. The cover carries a small "Built by Aardvark" credit; like the on-page Powered-by footer, it's removable with `poweredBy: false` on sites that use Aardvark's AI features.

Reuse the PDF instead of re-rendering every build

The whole-site PDF is the slowest part of a build, so you can let a build **reuse the copy already published** rather than rendering a fresh one every time. Give `pdf` as an object with `reuseForDays`:

```
pdf:
  reuseForDays: 7
```

On each build Aardvark fetches a tiny stable sidecar at `<baseUrl>/_aardvark/pdf-reuse.json` to find the last deployed fingerprinted PDF URL (falling back to the legacy `<baseUrl>/<slug>.pdf`, where the slug is derived from your `site.name`, e.g. "My Docs" → `/my-docs.pdf`). It then reads the build date embedded in that PDF itself (its `/CreationDate`). While that's **younger** than `reuseForDays`, the live PDF is republished as-is and the slow render is skipped; once it ages **past** the window, the next build renders a fresh one — whose new creation date restarts the clock. Because reuse republishes the exact bytes, the embedded date rides along unchanged, so republishing never resets the window — only a real render does. This needs a `baseUrl` to fetch from; anything missing or unreadable safely falls back to rendering, so a build never ends up without a PDF. Pass `--no-pdf-reuse`, or use a bare `pdf: true`, to always render.

The reuse fetch is locked to the `baseUrl` host (an SSRF guard — it won't follow a redirect to a different origin). So if your live PDF is served off-host (e.g. a redirect to an R2/CDN origin), reuse won't engage and the build just renders fresh; keep the PDF on the `baseUrl` host to benefit from reuse.

Password-protected pages: a fresh render drops protected pages from the PDF (their content would otherwise ship in the clear), but reuse republishes the live PDF as-is and can't re-apply that exclusion. So if you newly protect a page that was previously public, deploy once with `--no-pdf-reuse` (or `pdf: true`) — otherwise that page's content lingers in the public PDF until the

next fresh render (up to `reuseForDays` later). When the build reuses a PDF on a site that has protected pages, it notes this on the "Render PDFs" line (it isn't a warning — on a stable-protection site that would fire every build).

`_redirects` and alias stubs

Keep old URLs working after you move or rename a page. Two mechanisms feed the build — use either or both.

Per-page `aliases:` (front matter)

List a page's historical paths and each one forwards to the page:

```yaml
---
title: CLI reference
aliases:
  - /old-cli/          # forwards to /cli/
  - /legacy/cli.html   # an explicit file works too
  - from: /old-start/  # override the destination when moving a section
    to: "#quickstart"  # fragment-only values resolve against this page
---
```

String aliases forward to the page that owns them. Use the `from`/`to` mapping when an old URL should forward to a specific section or another URL; `to` accepts a root-relative URL, an absolute HTTP(S) URL, or a fragment such as `#quickstart` resolved against the owning page.

For every alias, Aardvark writes a tiny HTML stub at the old path with a `rel="canonical"` to the real page, `robots: noindex`, and an instant `<meta http-equiv="refresh">` (plus a JS redirect and a visible link). Google treats an instant refresh as a permanent move, and the canonical consolidates the old URL's ranking onto the new one — so search engines update themselves. Stubs work on **every** host and in `vark dev`, and each alias also gets a true `301` line in the `_redirects` file below.

Site-wide `redirects:` (config)

For rules that aren't tied to one page — including `*` wildcards and `:slug` / `:splat` placeholders — add a `redirects:` list to `aardvark.config.yaml`:

```yaml
redirects:
  - /blog/* /news/:splat 301        # a raw _redirects line, passed through as-is
  - from: /docs/:slug               # or the mapping form
    to:   /guide/:slug
    status: 301
```

Each list item is one rule on a single line (a multi-line value is truncated at the first newline). They're written verbatim to `_redirects`. A static generator can't expand a wildcard into files, so — unlike concrete `aliases:` — they produce **no stubs** and only take effect on a host that reads `_redirects` (Cloudflare Pages, Netlify, Vercel). On other hosts they're inert.

Tuning

```
aliases:
  htmlStubs: true       # write the per-alias HTML stubs (default true)
  redirectsFile: true   # also emit a _redirects file (default true)
  force: false          # append `!` to alias 301s — Netlify force flag (default false)
```

A bare `aliases: false` turns off stub generation entirely. One caveat for true `301`s: on Cloudflare Pages and Netlify a static file shadows a `_redirects` rule for the same path, so an alias's own stub wins over its `301` there. If you deploy only to those hosts and want header-level `301`s, set `aliases: { htmlStubs: false }` so the `_redirects` rule is unshadowed.

Deep nesting

This section nests **five levels deep** in the left sidebar — the deepest navigation in the sample site. It doubles as a worked example of how Aardvark assembles a sidebar tree and as a visual fixture for checking that deeply nested navigation still renders and indents correctly.

There is no depth limit. Each level is an ordinary page that points at its parent in front matter — children attach to a parent by its `id`, and every level below inherits the `docs` menu through the chain:

```
# this page (level 1) — the section root
id: deep-nesting
menu: docs

# a child page (level 2)
parent: deep-nesting
```

See Navigation menus for the full mechanism.

Walk down the tree — **Deep nesting → Level 2 → Level 3 → Level 4 → Level 5**. Open Level 2 to start the descent.

Level 2

You are two levels deep: **Deep nesting → Level 2**.

This page joins the tree with `parent: deep-nesting`, so it nests under the section root. Keep going to Level 3.

Level 3

You are three levels deep: **Deep nesting → Level 2 → Level 3**.

Halfway down. Continue to Level 4.

Level 4

You are four levels deep: **Deep nesting → Level 2 → Level 3 → Level 4**.

One more to go — Level 5.

Level 5

You are five levels deep: **Deep nesting → Level 2 → Level 3 → Level 4 → Level 5**. This is the deepest page in the sample site.

Sitting on this page, every ancestor in the sidebar is expanded and marked as the active trail, with this page highlighted as the current leaf — the same behavior the unit tests in `tests/test_theme.py` pin for arbitrary depth.

AI features

Aardvark is built for two audiences at once — the people reading your docs and the agents acting on them. The reader features add an assistant; the agent features make your site discoverable, queryable, and verifiable by automated clients. Everything is in one place here.

For readers

AI assistant

A built-in “Ask AI” chat panel for your readers — plus a full analytics dashboard (Top Questions, Coverage Gaps, and more) for you.

Cloud gateway

The managed metering proxy behind the built-in assistant — a prepaid balance with Stripe card-on-file and auto-top-up.

Build-time AI

Opt-in, cached OpenRouter features — generate frontmatter, example API responses, and Claude Code skills from your docs.

For agents

Self-hosting & MCP

Run `vark serve` to expose a live MCP server and serve each page as Markdown via content negotiation — the same tools work in the browser through WebMCP.

Agent readiness

A checklist for making your site agent-ready — what Aardvark publishes on every build, and the few steps only you can do.

Agent discovery

Always-on discovery endpoints — an MCP Server Card, OAuth/OIDC metadata, `auth.md`, and DNS-AID records — emitted by every build.

Web Bot Auth

Publish a key directory so your site can identify itself when an agent sends signed requests on its behalf.

More agent-facing output lives under Deploy

The files agents read most — `llms.txt`, `llms-full.txt`, and the Agent Skills index — are written by every build; see Generated files. The `vark ai-enrich` and `vark web-bot-auth-keygen` commands are in the CLI reference.

AI assistant & analytics

Aardvark ships a **native "Ask AI" assistant** for your readers, and behind it a full **conversation-analytics dashboard** that turns every question into product insight. It is a first-party feature — the assistant calls the Aardvark cloud gateway, which proxies the model, meters the spend, and records the conversation for analysis. There is no third-party widget to embed and no separate vendor to sign up with.

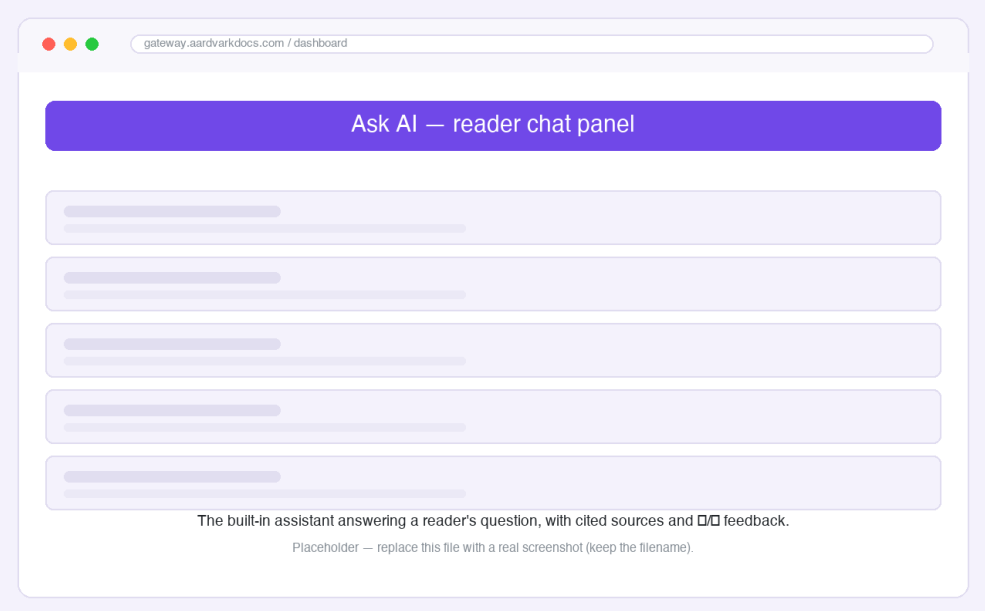
This page has two halves: the **reader's assistant** (what visitors see) and the **analytics dashboard** (what you see). The cloud gateway page covers the billing, metering, and Stripe setup that sit underneath both.

One assistant, one bill

The reader chat **and** every analytics model pass (clustering, per-turn analysis, the natural-language analytics assistant) are metered to **your** gateway account at the gateway's rate. You pay to analyze your own conversations; a depleted account simply pauses analysis rather than breaking. See How it's billed.

The reader's assistant

A floating **"Ask AI"** panel sits on every page. Readers ask a question in natural language; the assistant answers from **your content**, citing the pages it used, and they can rate each answer with / . It can read the whole corpus inline on the first turn (for a fast, zero-fetch answer) or navigate page-by-page, and it accepts **file attachments** (images, PDFs, text/code) when you leave them on.



The built-in assistant answering a reader's question, with cited sources and /

Placeholder — replace this file with a real screenshot (keep the filename).

Enable it

```
ai:
  assistant:
```

```
    enabled: true
    model: "~anthropic/claude-sonnet-latest"   # any model the gateway proxies; "latest"
alias needs the leading ~
```

The site also needs your **public** gateway key baked in as the `AARDVARK_KEY` environment variable at build time (`aardvark_live_…`, never the secret key — it ships in the static site). Provisioning a key and funding the account live on the cloud gateway page.

Options

All optional except `enabled`:

Key	Default	What it does
`enabled`	`false`	Master switch for the reader panel.
`model`	`~anthropic/claude-sonnet-latest`	The model the assistant answers with (must be vision/file-capable if you keep attachments on).
`gateway`	the managed gateway	Base URL of the gateway Worker (include the `/v1` suffix). Point it at your own gateway to self-host.
`reasoning`	model default	Reasoning control for a reasoning-capable model (`enabled` / `effort`).
`attachments`	on	Reader file uploads — see Reader attachments on the gateway page for the caps and cost notes.
`store_history`	`true`	Posts each finished turn to the gateway so it appears in your dashboard **and feeds the analytics below**.
`inlineContextMaxTokens`	model-derived	First-turn inline budget; `0` forces page-by-page navigation.

`prompt`	server-side	The system prompt is **not** baked into the public site — set it as the key's `system_prompt` when you mint the key.

store_history feeds the analytics

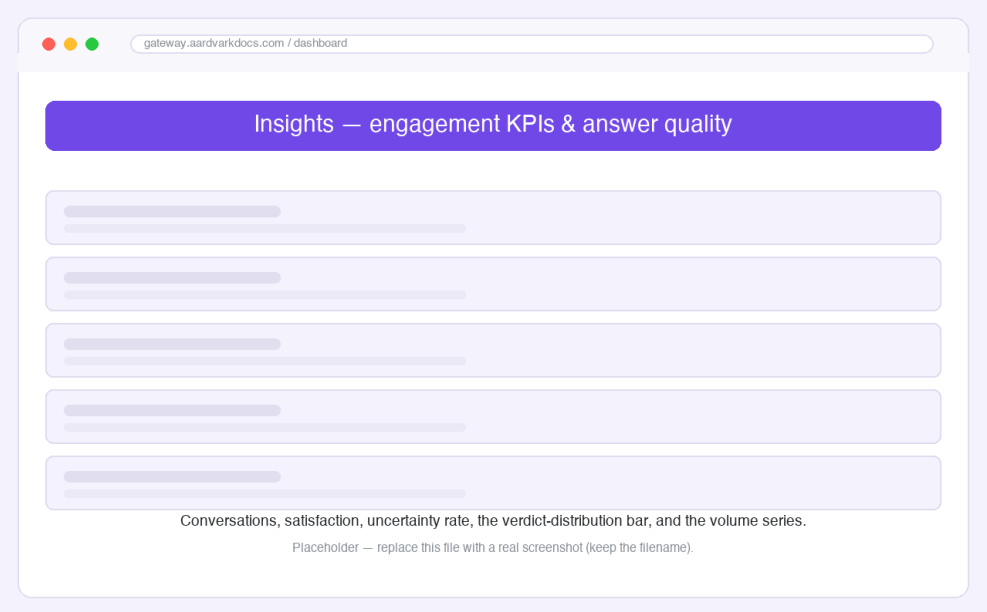
The entire analytics dashboard is built from **stored transcripts**. Leaving `store_history` on (the default) is what populates Insights, Conversations, Top Questions, and the rest. Set it to `false` and the assistant still works, but the dashboard has nothing to analyze.

The analytics dashboard

Every stored conversation feeds an analytics suite on the **gateway dashboard** — open `gateway.aardvarkdocs.com/dashboard` (or your own gateway host) and sign in via the **magic link** emailed to the account owner. (The dashboard is email-login only; the `aardvark_secret_…` key is the API/CLI credential — it authenticates the programmatic endpoints below, not the dashboard UI.) Cron passes on the gateway classify each answer, cluster the corpus into themes, and surface where your docs fall short.

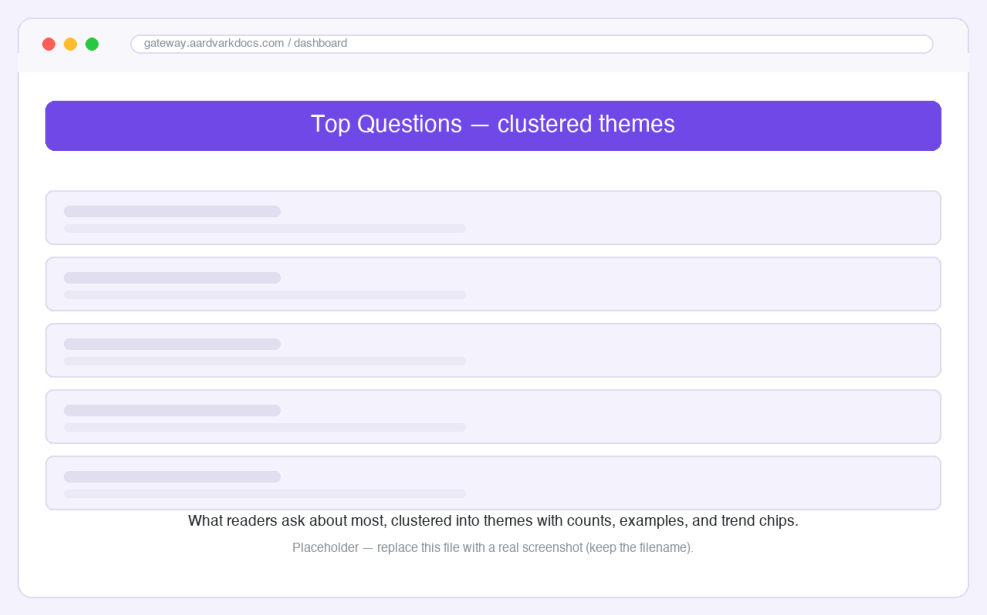
Insights — the overview

The **Insights** tab is a KPI overview over a 7 / 30 / 90-day window: total **conversations** and **answers**, the reader **satisfaction rate** (the share of votes), the **uncertainty rate**, an **answer-quality** distribution bar (every answer is graded `confident` / `unconfident` / `not_found` / `doc_gap` from the model's own reasoning trace), a **conversation-volume** series, the **language** mix, and a **support-deflection** rate.



Top Questions

A cron pass clusters recent conversations into broad **themes** — each a label, an approximate conversation count, example questions, and a trend chip versus the prior period. Use it to see what readers actually come for, and export the list to CSV.

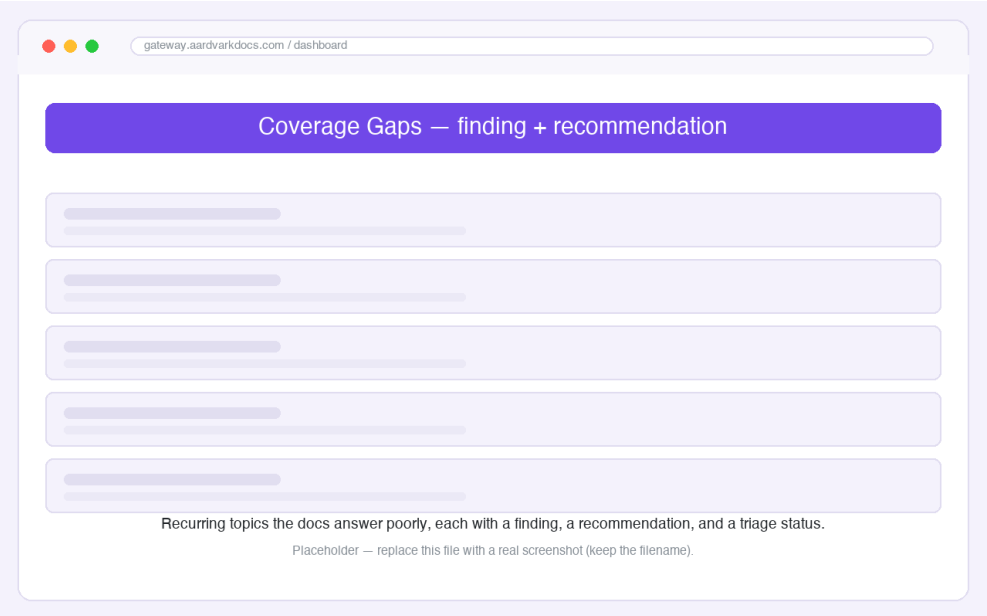


What readers ask about most, clustered into themes with counts, examples, and trend chips.

Placeholder — replace this file with a real screenshot (keep the filename).

Coverage Gaps

The companion pass clusters the **uncertain** answers — the questions your docs handle poorly — into recurring topics. Each gap card carries a **Finding** (what's missing or confusing) and a concrete **Recommendation**. A **Copy for LLM** button drops the finding + recommendation onto your clipboard to paste into an LLM or issue, you can set a **triage status**, and you can **export the gap as a ready-to-paste GitHub / Linear / Jira issue** — the status is keyed to a stable label hash, so it survives the next re-cluster.

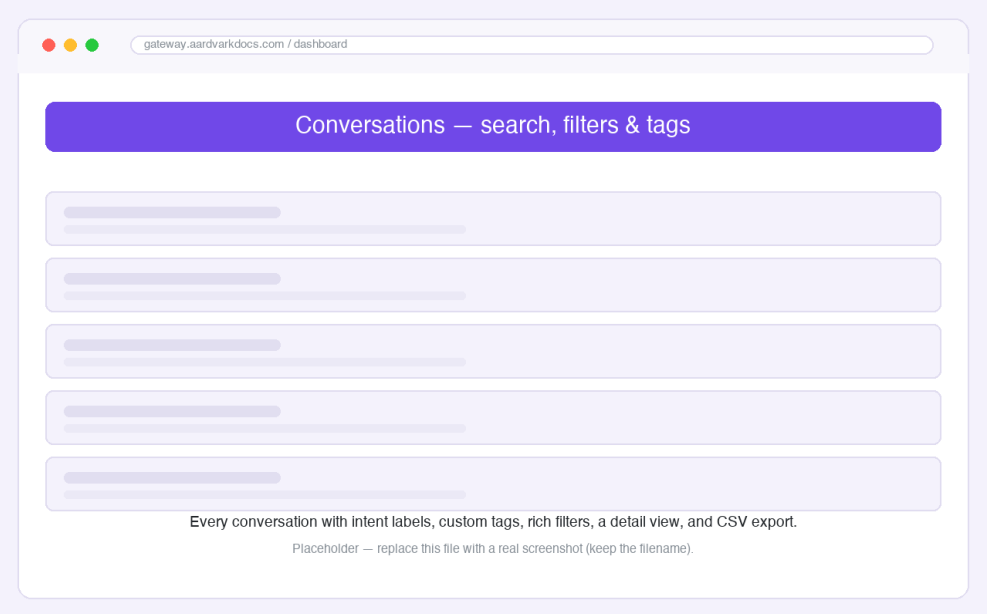


Recurring topics the docs answer poorly, each with a finding, a recommendation, and a triage status.

Placeholder — replace this file with a real screenshot (keep the filename).

Conversations

The **Conversations** tab is the raw record: every conversation with its per-turn **verdict**, an **intent** auto-label, and any **custom tags** you've defined (up to 20 per account — you give a tag a name and a description, and the analysis pass applies it, backfilling existing conversations). A rich filter bar narrows by full-text **search**, verdict, vote, intent, tag, and date range, and a **needs-attention** view surfaces the answers worth reading. Open any conversation for the full transcript, and **export** the filtered set to CSV (formula-injection-guarded).



Every conversation with intent labels, custom tags, rich filters, a detail view, and CSV export.

Placeholder — replace this file with a real screenshot (keep the filename).

Source Analytics

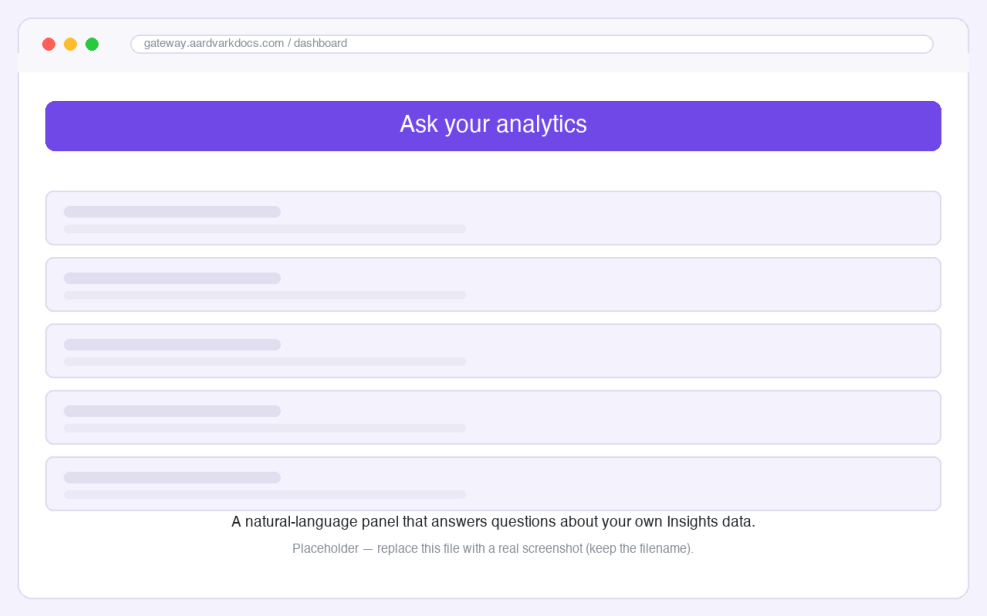
Which of your pages are doing the work: **most-cited sources**, plus the citations that show up disproportionately alongside **down-voted** answers — a signal that a popular page may be misleading rather than helpful.

Periods, trends & the email digest

Snapshots are versioned by **period** — week, month, or quarter — so you can navigate back through history and every cluster shows a **trend** against the prior period. Opt in to a **weekly or monthly email digest** (from the dashboard) and the gateway emails the account owner a summary of the latest Top Questions, new Coverage Gaps, and headline metrics.

Ask your analytics

A natural-language panel answers questions about your own Insights data — "what are readers most confused about this month?", "which pages get cited with thumbs-down?" — so you don't have to read every chart. Like the clustering passes it's metered to the account and rate-limited.



Programmatic export

For pulling analytics into your own pipelines, two dashboard-authed read APIs:

- **`GET /v1/activity`** — aggregate stats (conversations, answers, satisfaction, verdict distribution, deflection, languages) over a `?days=` window.
- **`GET /v1/threads`** — a keyset-paginated full-conversation export with `?since=` **incremental sync**, so a follow-up pull returns only what's new.

How it's billed

The reader assistant uses your configured `model` (a capable chat model by default). The **analytics** passes deliberately default to a cheap, large-context model (`~google/gemini-flash-latest`) so classifying and clustering a corpus stays inexpensive; an operator can override each pass independently. Every call — reader chat and analytics alike — rides the gateway's managed upstream key and is **metered to your account** at the published rate. Spend can't run away: each pass is bounded per cron tick and per account, and an account with no balance is skipped until it's funded again. Full pricing, funding, and Stripe details are on the cloud gateway page.

Privacy

The analytics are deliberately **leaner on identity than third-party tools**: there is no durable per-visitor id, so the dashboard reports **active conversations**, never a "unique users" count. Stored transcripts keep the **question and answer text** (to analyze), not attachment bytes, and they age out on the gateway's retention schedule along with everything derived from them (verdicts, tags, cited-source events).

Related

- Cloud gateway — the metering proxy behind the assistant: prepaid balances, Stripe card-on-file, auto-top-up, and reader attachments.
- Build-time AI — opt-in features that run during the build (frontmatter, example API responses, skills), also via OpenRouter.
- Self-hosting & MCP — running a built site, and serving it over MCP.

Aardvark cloud gateway

The **Aardvark cloud gateway** is the managed metering proxy that sits behind the built-in **"Ask AI"** assistant. The assistant in a reader's browser never talks to a model directly — it calls the gateway, which proxies the request to the model behind Aardvark's managed keys, **meters** the spend at Aardvark's published per-model rates (subscription plans meter at a member discount), and **cuts off** an account that has run out of credit.

This page is conceptual and how-to, for both **customers** (whose docs site uses the gateway) and **operators** (who run a gateway). For end-to-end deployment and Stripe setup, see `gateway/DEPLOYMENT.md` in the gateway source.

> The gateway is **optional**. A site can point the assistant at your own OpenRouter key, or skip the assistant entirely. The gateway is what you reach for when you want **metering, per-customer balances, and billing** in front of the model.

The prepaid model

Every account holds a **prepaid balance**. On the **Free / pay-as-you-go** tier, each metered chat request **debits** the balance at the published per-model rate; when the balance is exhausted, the gateway stops serving that account's **paid** chat requests until it is funded again — so spend can never run away past what the account holds. (A `:free` model always answers at $0 and never touches the balance.)

Subscribers add a monthly **included-AI grant** on top of this. A metered request drains the grant first (any rollover, then the current period's grant) and only then — per the account's **overflow policy** — the prepaid balance:

- **Cap-and-hold** (the default): when the grant is spent, paid AI **pauses** and readers fall back to `:free` answers **even while the prepaid balance is still funded** — so the bill stays predictable. Adding funds does **not** resume paid AI here; the next period's grant (or a plan change) does.
- **Fall back to your balance** (opt-in): overflow draws the prepaid balance at the subscriber's member rate, up to a customer-set **cap**, then pauses like cap-and-hold.

So for a subscriber, "out of funds" and "included AI used up" are **distinct** states with different recovery actions — the dashboard says which one an account is in.

An account is funded in one of two ways:

- **Operator top-up** — the operator adds credit directly (an admin action). Always available, and the only option when Stripe is not configured.
- **Customer self-funding** — the customer stores a card and funds their own balance, described next.

Customer self-funding (card-on-file + auto-top-up)

When the operator has enabled Stripe, a customer manages their own funding from the **Billing & Auto Top-Up** section of the gateway **dashboard** (open `/dashboard` on the gateway and sign in via the magic link emailed to the account owner — the dashboard is email-login only).

Dashboard roles. Team accounts have three roles: **owner**, **admin**, and **member**. Owners and admins see the **Billing** tab; plain members do not, and direct Billing links fall back to the dashboard overview. Billing reads and mutations — saved card, auto-top-up, manual top-ups, and the self-serve lift for a billing-driven pause — are enforced server-side for owners and admins only.

Store a card. Adding a card sends the customer to **hosted Stripe Checkout** — a full-page redirect to Stripe's own form. Card details are entered on Stripe, never on the dashboard; the gateway stores only a token for the saved card plus the display brand / last-4 / expiry. There is no card form embedded in the page.

Enable auto-top-up. With a card on file, the customer turns on auto-top-up by choosing two numbers:

- an **amount** to charge per top-up, and
- a **trigger threshold** — the balance level that fires a top-up.

The threshold must be **below** the amount, so each top-up lifts the balance back above the line. When the balance crosses the threshold, the gateway charges the saved card the chosen amount and credits the balance — off-session, with no reader interaction.

Top up now. The same section has a manual **Top up now** button for a one-off charge of the saved card.

Safety properties

Auto-top-up is built to be safe to leave on:

- **Charged at most once per low-balance episode.** Overlapping triggers — or a manual Top up now firing at the same moment — cannot double-charge the card.
- **Self-disables after repeated declines.** A consistently failing card stops being retried on every crossing, and the customer is alerted to fix it.
- **Whole-cent amounts, with a minimum** (a **$10** floor by default).
- **Durable crediting.** A captured charge credits the balance exactly once, even if something fails between the charge and the credit.

Removing the card (also from the Billing section) disables auto-top-up.

For operators: enabling Stripe

The gateway runs as a prepaid-only metering proxy out of the box. Customer self-funding is **off until you configure Stripe**, and turning it on is, in short:

1. Set two Worker secrets — `STRIPE_SECRET_KEY` and `STRIPE_WEBHOOK_SECRET`.

Create a Stripe **webhook** pointing at `…/v1/stripe/webhook`, subscribed to all seven events the gateway handles — the two PaymentIntent ones that back top-ups, plus the five subscription-lifecycle events that back the monthly plans:

```
payment_intent.succeeded          payment_intent.payment_failed
customer.subscription.created     customer.subscription.updated
customer.subscription.deleted
invoice.paid                      invoice.payment_failed
```

The webhook is the durable credit backstop. If you already had an endpoint from before the subscription work, **edit it to add the five subscription events** — without them plan invoices never flip an account to `past_due` or recover it, and a Stripe-side cancel won't downgrade the account until the grace sweep catches up.

3. Apply the gateway's payment **database migrations** before deploying the updated Worker.

With Stripe left unconfigured the gateway behaves exactly as before — operator top-ups only. Full steps, optional tuning, and a test-mode walkthrough are in `gateway/DEPLOYMENT.md`.

Reader attachments

Readers can attach **files** to a question — images, PDFs, and text/code/markdown — from the assistant's composer (a paperclip button, drag-and-drop onto the panel, or paste). This is controlled by `ai.assistant.attachments` and is **on by default**.

```yaml
ai:
  assistant:
    enabled: true
    model: "~anthropic/claude-sonnet-latest"   # must be vision/file-capable (see below)
    attachments:
      enabled: true        # master switch — `attachments: false` is the shorthand to turn
it off
      maxFiles: 4          # per-message file cap (default 4)
      maxFileSizeMb: 10    # per-file size cap in MB (default 10)
      pdfEngine: pdf-text  # PDF parsing: pdf-text (default) | mistral-ocr (scanned) |
native
```

Beyond the per-file cap, the picker also enforces a **combined** image/PDF budget (base64-encoded) that matches the gateway's hard per-request limit, so it rejects a set that would always be refused upstream rather than failing mid-send. (Text attachments are inlined as text and don't count toward that budget.)

How each kind is sent to the model:

- **Images** are sent as image content (base64) for the model to view.
- **PDFs** are sent as a file part with a parsing plugin; `pdfEngine: pdf-text` extracts the text cheaply, while `mistral-ocr` is for **scanned** PDFs (image-only pages) and costs more.

- **Text / code / markdown / CSV / JSON** are read in the browser and inlined as text — the cheapest path, and it needs no special model support.

Model capability

The assistant uses one **build-time** model (`ai.assistant.model`). Attachments only work if that model can accept them — point it at a **vision/file-capable** model (most current Claude and GPT models qualify). A model that can't will **error or ignore** the attachment, so choose the model deliberately when you leave attachments on.

Cost

Attachments **spend more**. Images and PDFs cost far more tokens than the equivalent text, and a large file can dominate a turn's cost. Because attachments are on by default, budget for the extra spend — or lower the caps, narrow `accept`, or set `attachments: false`. The gateway's spend cap remains the hard ceiling: an attachment-aware reserve holds a larger amount up front for a turn that carries files, and the true cost is settled when the turn completes.

> To keep follow-up turns cheap, an attachment's data is sent **only on the turn it's added** — the file isn't replayed on later questions in the same conversation. The model can't re-inspect a file on a follow-up; re-attach it if you need it again. (Rehydrated history after a page navigation keeps the file's name only, not its contents.)

Security & privacy

- **Untrusted content / prompt injection.** A file's contents (including extracted PDF text or OCR) enter the model's context and can carry injected instructions. This is inherent to letting readers supply content; treat assistant answers accordingly.
- **Privacy.** Attachments are sent to the model provider (via the gateway to OpenRouter) to answer the question. Stored chat transcripts keep the **question text only**, not attachment bytes.
- **Payload limits.** The client caps file count and size; the gateway also enforces per-request count and byte limits, so an oversized upload is rejected rather than billed.

Related

- AI assistant & analytics — enabling the built-in Ask AI panel for readers, and the operator analytics dashboard (Insights, Conversations, digests) it feeds.
- Content Reach — which sections of each page readers reach and which controls they use, on the Business and Enterprise plans. Carries no spend: it is telemetry, not metered AI.
- Build-time AI — opt-in features that run during the build, also via OpenRouter.
- Self-hosting & MCP — running a built site (and its MCP server) yourself.

Build-time AI

Aardvark has optional, build-time AI features that run through the metered **Aardvark cloud gateway**, so you can use **any model it proxies**. They are **off by default**, run only when explicitly enabled, and their results are **cached** by content hash in `.aardvark-cache/ai/` — so unchanged content never re-calls the API and builds stay fast and deterministic.

Enable

1. Set a key (the AI features are built into the binary — nothing extra to install):

```
export AARDVARK_SECRET_KEY=aardvark_secret_...
```

Turn on the features you want in `aardvark.config.yaml`, and pick a model:

```
ai:
  frontmatter: true   # generate missing description + keywords
  examples: true      # example API responses for OpenAPI pages
  skills: true        # plan + generate skills/ (on demand, via vark ai-enrich)
  model: "~anthropic/claude-sonnet-latest"   # any model the gateway proxies; "latest" alias
needs the leading ~
```

Any run can override the configured model with `--model <slug>` (e.g. `vark build --model "~anthropic/claude-sonnet-latest"`). If the key or the SDK is missing, the features simply no-op.

What each does

- **`frontmatter`** — for pages missing `description`/`keywords`, generates them and fills the page metadata (used in `<meta>` tags and `llms.txt`).
- **`examples`** — generates a realistic example response for each OpenAPI operation and shows it on the reference page.
- **`skills`** — plans a set of Claude Code "skills" from your docs, then generates a full `SKILL.md` for each (grounded in the pages it cites) under `skills/<name>/` in your project root. Runs **only** via `vark ai-enrich` (below), not during `vark build`. `vark build` then publishes whatever is in `skills/` as an Agent Skills Discovery index at `/.well-known/agent-skills/index.json`.

Run on demand

`vark ai-enrich` runs enrichment outside a full build — it writes generated `description`/`keywords` back into your source Markdown frontmatter and, if `ai.skills` is on, (re)generates the `skills/` directory. Skills live outside the build output, so a normal `vark build` never regenerates or wipes them — they're refreshed only when you run `ai-enrich`.

Docs Quality Checks

Docs Quality Checks connect your GitHub repository to the **Aardvark cloud gateway** and run Aardvark's built-in **AI-powered docs workflows** for you — on a **schedule**, on **every commit**, or on demand — then opens a **pull request** with the changes for you to review and merge.

You set it up entirely from the gateway **dashboard**: connect a repo, tick the capabilities you want, choose when they run, and watch the run history. There's nothing to install in your repo and no workflow YAML to write — the runs happen on Aardvark's own runners.

> Docs Quality Checks run the **same** capabilities as Build-time AI and the `vark author` command, just unattended in CI. It's the hands-off way to keep metadata, skills, translations, and prose fresh as your docs change.

How it works

Everything runs on **Aardvark's side** — your repository carries no workflow files and no secrets:

1. You connect a repo and pick capabilities in the dashboard.
2. The gateway stores your choices and schedule. **Nothing is written to your repo.**
3. When an automation is due (schedule, a push to the chosen branch, or Run now), the gateway runs it on **Aardvark's own runners**: it checks out your repo with a short-lived, repo-scoped token, runs the `vark` capability against your docs, and opens or updates a pull request on an `aardvark/<capability>` branch.
4. The run's status — and its metered cost — flow back to the dashboard as **per-capability run history**.

Because the real `vark` CLI does the work, nothing about how a capability works — its prompts, its logic — ever leaves the binary. The gateway schedules the runs, opens the PRs, and meters the spend.

Connect a repository

1. Open the gateway **dashboard** (`/dashboard`) and go to **Docs Quality Checks**.
2. Click **Connect GitHub**. You'll be sent to GitHub to install the Aardvark app and **choose which repositories** to grant it — pick only the repos you want automated.
3. Back in the dashboard, each connected repo appears with the capabilities you can enable.

> You stay in control of access on GitHub: the app sees only the repositories you select, and you can change or revoke that at any time from your GitHub settings.

Choose checks and a schedule

For each repo, turn on the checks you want and pick when each one runs:

- **Run on every commit** — the check runs on each push to the branch.
- **On a schedule** — enter a 5-field cron expression (e.g. `0 9 * * 1` for Mondays at 09:00 UTC).
- **Run now** — trigger a one-off run any time from the dashboard.

A check can use either trigger, both, or neither (manual-only). Click **Save** — this only updates your automation config in the gateway; nothing is committed to your repo.

Scope a run

By default a check runs over **every page**. The three page-rewriting checks — Apply style guide, Regenerate keywords, and Refresh descriptions — can instead run over just part of your docs:

- **Diff only (changed pages)** — a checkbox under **Run on every commit**. When on, a push restyles only the pages that push changed, not the whole docset (and opens no PR when a push touched no docs). It applies to **pushes only**: a **scheduled** run has no diff to work from, so it always covers every page. One more case is skipped by design: the **first push that creates a branch** has no "before" commit to diff against, so a diff-only automation doesn't run on it (rather than silently restyling everything) — the next ordinary push to that branch is scoped normally.
- **Run now → path** — the **Run now** control has an optional **path** field. Fill in a file or a directory to scope that one run to it; leave it blank to run over everything.

Scoped runs **build on the open PR** rather than replacing it. Aardvark opens one pull request per check (on an `aardvark/<check>` branch); when a later scoped run touches only a few pages, it **keeps** the styling the PR already proposed for every other page and just updates the pages it restyled — so the PR accumulates across commits instead of shrinking to the latest run's pages. Each run also rebases the PR on your current branch, so the proposed changes stay current. (A **full** run — a scheduled run, or any run over every page — regenerates the whole PR from scratch.)

From the CLI, the same two ways to scope a run:

```
# A file or a whole directory (repeatable)
vark author --action styleguide --yes --rulesets google --path guide/

# Only the pages this branch changed vs another ref (e.g. main)
vark author --action keywords --yes --diff main
```

`--diff <ref>` restyles only the pages whose source changed versus that git ref — the same mechanism the dashboard's Diff only toggle uses on a push. `--page` is the **single-page** selector — note it differs from `--path` on a trailing slash: `--page guide/` selects the `guide.md` page, whereas `--path guide/` selects the whole `guide/` subtree (and `--path` is repeatable). Don't combine `--path`/`--page` with `--diff` or `--all`; with none of them, the check runs on every page.

`--diff` selects pages by their **own** source changing — including a changed reusable partial (styleguide only). It doesn't follow includes: a change confined to a data file (or, for keywords/descriptions, a partial) won't re-run the pages that merely consume it.

The checks

Every check runs the **same `vark` capability** two ways: unattended from the **dashboard**, where it opens a pull request, or locally from the **command line**, where it edits your working tree directly. Use the dashboard for hands-off, reviewed automation; use the CLI to run a check on demand or wire it into your own pipeline.

The CLI checks need your gateway **secret key** (the same account that powers the dashboard):

```
export AARDVARK_SECRET_KEY=aardvark_secret_...
```

The `vark author` checks **preview a diff by default** — add `--yes` to write the changes. They run on **every page** unless you scope the run with `--path <file-or-dir>` or `--diff <ref>` (or `--page <path>` for one), and `--model <slug>` overrides the configured model. The summary table:

Check	Dashboard label	CLI
Apply style guide	Apply style guide	`vark author --action styleguide --yes --rulesets <id,id,…>`
Enrich metadata & skills	Enrich metadata & skills	`vark ai-enrich`
Regenerate keywords	Regenerate keywords	`vark author --action keywords --yes`
Refresh descriptions	Refresh descriptions	`vark author --action description --yes`
Translate documentation	Translate documentation	`vark build --translate`

Apply style guide

Rewrites each page's **prose** to follow one or more **style rulesets**, in **precedence order** — without changing meaning, code, commands, front matter, link targets, or `{% %}` template blocks. It also restyles your reusable **partials** (_-prefixed include-only files) under your base `content/`, so shared prose follows the style guide too — the other checks act on routable pages only. (Partials that live under a translated content directory aren't included; on a multilingual site, style the base-language ones.) Aardvark ships six rulesets, condensed from each guide's own source of truth:

Ruleset	id	Focus
Microsoft Style Guide	`microsoft`	Microsoft's technical-writing conventions — warm, clear, concise.

Google Style Guide	`google`	Google developer-docs style — second person, active voice, sentence case.
Alex (considerate writing)	`alex`	Inclusive, considerate language — flags insensitive or exclusionary wording.
Writegood	`writegood`	Common prose weaknesses — passive voice, weasel words, wordiness.
Proselint	`proselint`	A broad prose linter — clichés, redundancies, jargon, malformed dates.
Readability	`readability`	Plain language — shorter sentences, simpler words, tighter paragraphs.

Precedence matters. The order is meaningful: a ruleset **higher in the list wins** when two give conflicting guidance; otherwise every selected ruleset applies with equal force. You must pick **at least one** — there is no implicit default, so an unconfigured style-guide check never runs (it would otherwise silently restyle against a guide you never chose).

In the dashboard: enable Apply style guide, then use the **ruleset multi-select** to add rulesets and the **↑ / ↓ arrows** to order them by precedence (slot 1, the violet badge, wins conflicts). Save stays disabled until at least one ruleset is picked. Then schedule the check or hit **Run now**.

From the CLI: pass the ids comma-separated, **highest precedence first**:

```
# Preview the changes on every page (Microsoft wins conflicts, then Google)
vark author --action styleguide --rulesets microsoft,google

# Apply them
vark author --action styleguide --yes --rulesets microsoft,google

# Style a single page against all six
vark author --action styleguide --page docs/intro.md --yes \
  --rulesets microsoft,google,alex,writegood,proselint,readability
```

Omitting `--rulesets` (or passing an unknown id) fails fast and prints the six valid ids — the same selection the dashboard enforces. Combining many rulesets makes a large prompt (~26K tokens for all six); the run warns when the combined prompt is large, and a smaller model may not fit it — prefer the fewest rulesets that cover your needs.

Enrich metadata & skills

Fills in any missing page `description`/`keywords` front matter and, when skills are enabled, (re)generates the `SKILL.md` files for your planned agent skills.

In the dashboard: enable Enrich metadata & skills, then schedule it or **Run now**.

From the CLI (driven by the `ai:` block in `aardvark.config.yaml` — see Build-time AI):

```
vark ai-enrich
```

Regenerate keywords

Refreshes the SEO `keywords` front matter on every page from its current content.

In the dashboard: enable Regenerate keywords, then schedule it or **Run now**.

From the CLI:

```
vark author --action keywords --yes      # every page; drop --yes to preview
```

Refresh descriptions

Rewrites each page's one-sentence `description` — the text used in search results and social cards.

In the dashboard: enable Refresh descriptions, then schedule it or **Run now**.

From the CLI:

```
vark author --action description --yes    # every page; drop --yes to preview
```

Translate documentation

Translates pages that are new or changed since the last run into each configured target language.

In the dashboard: enable Translate documentation, then schedule it or **Run now**.

From the CLI (needs `i18n` target languages configured — see Build-time AI):

```
vark build --translate
```

> `vark author --list-actions` prints the checks runnable headless. These are the same capabilities documented under Build-time AI — the dashboard just runs them unattended in CI.

Review the results

Every run opens a pull request on an `aardvark/<capability>` branch (re-runs update the same PR), so **nothing lands on your default branch without your review**. Merge it if the changes look good, or close it. The dashboard's **run history** links straight to each run and its PR.

Cost

Each run **spends from your gateway balance** on two things, both shown per run in the dashboard:

- **AI** — the capability's model calls, metered exactly as a local `vark build` / `vark ai-enrich` would be.
- **Compute** — because the run executes on Aardvark's runners (not your own GitHub Actions), the runner minutes are metered too and billed to your balance.

Schedule the heavier capabilities (translation, or a style-guide pass over a large site) accordingly, and keep an eye on your balance from the cloud gateway dashboard. A run is skipped (and shown as such) if your balance can't cover it — top up to resume.

For operators

Docs Quality Checks are part of the cloud gateway. Offering them to your customers means installing a GitHub App with the right permissions and webhook and setting a few Worker secrets; the full setup — permissions, subscribed events, and the recommended OAuth identity check — is in `gateway/GITHUB_INTEGRATION.md` in the gateway source. With the app unconfigured, the dashboard simply shows that Docs Quality Checks aren't available, and the gateway runs exactly as before.

Related

- Build-time AI — the same capabilities, run during a local build.
- Aardvark cloud gateway — the metered account that powers and bills these runs.
- Agent readiness — what fresh metadata, skills, and clean prose buy you.

Self-hosting & MCP

Deployment covers handing the static `build/` directory to a host (Cloudflare Pages, Netlify, S3, …). This page covers the other option: **running the site yourself** with `vark serve` — a single hardened process that serves the static site and a live **MCP server**, so AI clients can query your docs as a tool — and packaging it as a container ready for the open internet.

`vark serve`

`vark serve` is the production counterpart to `vark dev`. It does **not** build, watch, or live-reload — run `vark build` first, then:

```
vark build
vark serve                      # serve ./build on 0.0.0.0:8080, with /mcp
vark serve --port 80 --no-mcp   # static only, no MCP endpoint
```

It serves the build exactly as a CDN host would — honoring the `_headers` and `_redirects` the build emits (see Generated files), so content types, the `.md`-as-`text/plain` rule, redirects, and the `ai-config.json` `no-store` rule all match. That's the point of serving from the same tool that built the site: there's no second web-server config to drift from what the build produced. It also serves the Web Bot Auth key directory at its extension-less well-known path — which a generic static resolver would mistake for an HTML page — when you've enabled it.

The serve stack (uvicorn/starlette/MCP) ships with every Aardvark install — `pip install aardvark`, the standalone binary, and the Docker image below — so there's nothing extra to add.

Markdown for Agents

`vark serve` also does Markdown-for-Agents content negotiation automatically: request any page with `Accept: text/markdown` and you get that page's Markdown — the same per-page `.md` the build already emits (Generated files) — at the **same URL** as `text/markdown; charset=utf-8`, while browsers keep getting HTML. The response also carries estimated token counts (`x-markdown-tokens` / `x-original-tokens`) and `Vary: Accept`, so a cache in front keeps the HTML and Markdown variants separate. There's nothing to switch on — it's automatic whenever per-page `.md` files are published (the default):

```
curl -H 'Accept: text/markdown' https://your-docs.example.com/guide/intro/
```

Put a CDN in front

`vark serve` is one process designed to sit **behind a CDN** (Cloudflare, Fastly, CloudFront). The CDN terminates TLS, caches the static tier, and is your real defense against volumetric abuse — so the origin mostly serves cache fills and a single replica comfortably handles a site with hundreds of thousands of monthly readers. Scale out by running more replicas (each is stateless); the `/mcp` endpoint is dynamic and is not CDN-cacheable.

If you'd rather front it with your own nginx/Caddy, you can — but then you own translating `_headers`/`_redirects` into that server's config and keeping it in sync with each build. Serving through `vark serve` avoids that entirely.

The Docker image

Drop the multi-stage `Dockerfile` from this project's root into your own docs project (next to `aardvark.config.yaml`), then build and run it:

```
DOCKER_BUILDKIT=1 docker build -t my-docs .
docker run --rm -p 8080:8080 --read-only --tmpfs /tmp my-docs
```

The build stage downloads the published `vark` release binary (no source build, no Python/uv toolchain) and uses Node 22 only to bundle the islands. It **bind-mounts** your project read-only and renders it in-container, so your Markdown never lands in an image layer — only the built `build/`, the `vark` binary, and your config travel in the final runtime image. That image runs as a non-root user and is read-only-rootfs friendly (give the self-extracting binary a writable `/tmp` via `--tmpfs /tmp`), with a `/healthz` health check. Build multi-arch with `docker buildx build --platform linux/amd64,linux/arm64`. Behind a CDN/LB that terminates TLS, point the container's rate limiter at it (see below):

```
docker run -p 8080:8080 --tmpfs /tmp my-docs --trusted-proxy 173.245.48.0/20
```

By default the image builds with the **latest published `vark` release**, so each rebuild picks up the newest version automatically. To pin a specific version instead, pass `--build-arg VARK_VERSION=X.Y.Z` (any release that includes `vark serve`, i.e. `>= 0.1.6`). For a tamper-checked build, also pass `--build-arg VARK_SHA256=<hash>` to verify the downloaded binary against the per-architecture hash published with that release; left unset, the build proceeds on HTTPS trust alone and warns.

The MCP server

When `/mcp` is enabled (the default), the server exposes a stateless Model Context Protocol endpoint over Streamable HTTP that wraps your docs as four read-only tools:

Tool	What it returns
`search_documentation`	Ranked pages for a query (same scoring as the on-page search box)
`fetch_document`	One page's raw Markdown
`list_documentation`	The navigation index — every page's title, path, headings, glossary
`get_full_corpus`	The whole site as one Markdown document (size-guarded)

These wrap the artifacts the build already emits (`search-index.json`, `metadata.json`, the per-page `.md` files, `llms-full.txt`). On a site with the on-page search box or the AI assistant enabled, those exist already. For an **MCP-only** site (neither enabled), set `mcp: true` in `aardvark.config.yaml` so the build still writes the full-text index and navigation corpus:

```
mcp:
  enabled: true
```

Connecting a client

Point any MCP client at `https://your-docs.example.com/mcp`:

- **Claude.ai / Claude Desktop** — add a Custom Connector with that URL.
- **Claude Code** — `claude mcp add --transport http my-docs https://your-docs.example.com/mcp`
- **IDEs (Cursor, etc.)** — add an HTTP MCP server pointing at `/mcp`.

A `GET /mcp` returns `405` by design (the server pushes nothing); clients use `POST`.

WebMCP — the same tools, in the browser

The same four tools are also exposed to **in-browser AI agents** (Chrome's built-in agent, extensions) via WebMCP: every page calls `navigator.modelContext` on load and registers `search_documentation`, `fetch_document`, `list_documentation`, and `get_full_corpus`. They run entirely client-side — same-origin fetches over the artifacts above, ranked with the on-page search box's scorer — so this works on **any** host (a CDN, static hosting, `vark serve`), not only the MCP server.

WebMCP rides the same `mcp: true` switch — it's the browser transport of the same tool set, so there's nothing extra to turn on. It's a progressive enhancement: in a browser without WebMCP, or over plain HTTP (the API is secure-context-only — HTTPS or `localhost`), the script is a silent no-op. Keep `markdownMenu` on (the default) so `fetch_document` / `list_documentation` have the per-page `.md` files and `metadata.json` to read.

To run the `/mcp` server **without** injecting the in-browser client (for example, under a strict Content-Security-Policy), opt out while keeping the server on:

```
mcp:
  enabled: true
  webmcp: false
```

Under a strict `script-src 'nonce-…'` CSP, the small inline config script — emitted **only** when you override a search default (`search.compress: false` or a custom `search.ranking`) — is blocked; the client then falls back to its built-in defaults (gzipped index, default ranking), so the tools still work but a custom ranking wouldn't reach them. A default-search site emits no inline script at all. Use `mcp.webmcp: false` to drop the browser client entirely.

Raw event attributes passed through component `attr`, such as `attr={'onchange': '...'}`, are compiled into event listeners during island hydration. If your CSP permits those handlers,

`script-src` must allow `unsafe-eval`; a strict Trusted Types policy can block them outright. For strict-CSP deployments, prefer a small snippet island or built-in component callback over raw `on*` attribute handlers.

Hardening notes

Read these before exposing the server to the open internet.

- **Set `--trusted-proxy` behind a CDN.** The per-IP rate limit on `/mcp` keys on the client IP. Behind a CDN the direct peer is the CDN, so pass its CIDR(s) with `--trusted-proxy` (repeatable) — then the limiter reads the real client from `CF-Connecting-IP` / `X-Forwarded-For`. Without it, forwarded headers are ignored (they're spoofable from an untrusted peer) and **every client shares one bucket**. `vark serve` prints a warning when no trusted proxy is configured.
- **The in-process limiter is a backstop, not a DDoS shield.** It caps a single abusive client and a runaway agent loop on one replica; it does not coordinate across replicas. Real volumetric defense is the CDN in front.
- **TLS is the CDN/LB's job.** `vark serve` speaks plain HTTP; terminate TLS at the edge.
- **Private docs: gate the whole container.** `/mcp` is open by design — it only exposes the same `.md` and `metadata.json` files already fetchable from the static site, so gating `/mcp` while leaving the site public is security theater. For private docs, put the entire container behind your CDN access rules or an SSO proxy.
- **Rotated keys propagate.** `/_aardvark/ai-config.json` (which carries the rotatable assistant key) is served `no-store` straight from the build's `_headers`, so a rotation isn't held stale at the edge.

Agent readiness

isitagentready.com (Cloudflare's Agent Readiness Scanner) grades how well a site cooperates with AI agents: can they discover it, read it, learn the rules, and find its machine interfaces? Aardvark publishes **almost all** of those signals automatically on every `vark build` — this page is the checklist, so you can see what's already handled and do the handful of steps only you can do.

How the score works

The scanner runs a set of named checks grouped into five categories and reports a **level** (the top rung is Agent-Native). Four categories count toward your level:

Category	What it asks
Discoverability	Can an agent find your content and interfaces? (`robots.txt`, `sitemap.xml`, `Link` headers, DNS-AID)
Content accessibility	Can it read your pages as clean text? (Markdown negotiation)
Bot access control	Have you stated the rules for bots/AI? (`robots.txt` AI rules, Content Signals, Web Bot Auth)
Discovery	Can it find your APIs, auth, MCP server, skills? (API catalog, OAuth/OIDC, `auth.md`, MCP card, A2A card, agent skills, WebMCP)

A fifth category, **Commerce** (x402, MPP, UCP, ACP, AP2), is reported but **does not count toward your score** — it only applies to shopping/checkout sites, so a docs or marketing site can ignore it.

What Aardvark publishes for you (no work required)

Most are emitted on **every build** with no configuration — set `baseUrl` in `aardvark.config.yaml`, deploy over HTTPS, and they pass. A few are conditional, as the Notes column calls out: WebMCP rides on `mcp: true` (MCP is opt-in), the API catalog appears once you embed OpenAPI specs, and the Agent Skills index lists whatever `skills/` you ship.

Check	Endpoint / signal	Notes
robots.txt	`/robots.txt`	Includes wildcard rules that apply to AI bots
Content Signals	`/robots.txt` (`Content-Signal:`)	Declares search / ai-input / ai-train usage; tune under `robots.contentSignals`

Sitemap	`/sitemap.xml`	Every page, with `lastmod`
Link headers	`Link: …; rel="service-desc" / "describedby" / "api-catalog"`	Added by `vark serve` and the generated `_headers`
Markdown negotiation	`/<page>.md` + “View as Markdown”	Agents fetch the clean Markdown of any page
MCP Server Card	`/.well-known/mcp/server-card.json`	Advertises the MCP server `vark serve --mcp` hosts
WebMCP	`/_aardvark/webmcp-<sha>.js`	In-page tools agents can call — **opt-in**, requires `mcp: true`; generated pages are rewritten to the fingerprinted file
OAuth / OIDC discovery	`/.well-known/oauth-authorization-server`, `/.well-known/openid-configuration`	See “Point OAuth at your IdP” below
OAuth Protected Resource	`/.well-known/oauth-protected-resource`	RFC 9728
auth.md	`/auth.md`	Human + agent summary of how to authenticate
Agent Skills index	`/.well-known/agent-skills/index.json`	Lists every `skills/<name>/SKILL.md` you ship
API catalog	`/.well-known/api-catalog.json`	Published when your site embeds `{% openapi %}` specs (see OpenAPI)
llms.txt	`/llms.txt`, `/llms-full.txt`	Bonus agent-friendly content index

See Agent discovery and Generated files for the details of each. The takeaway: out of the box, with just a `baseUrl`, an Aardvark site already clears most of the scanner.

Hand this site to an agent (the page menu)

Discovery files help an agent that already found your site. To let a **reader** point their own agent at it, the "View in Markdown" menu at the top of every page also offers one-click hand-offs (all on by default, hidden per item under `markdownMenu`):

Menu item	What it does	Works in
Copy MCP Server	Copies this site's MCP endpoint URL	Claude Code (`claude mcp add --transport http`), Claude Desktop / claude.ai (Customize → Connectors → Add custom connector) — shown only when `mcp: true`
Install Skill	Downloads an Agent Skill (`SKILL.md`) that teaches an agent how to search this site	Claude Code (drop in `~/.claude/skills/`), Claude apps (Customize → Skills → Upload a skill)
Install Plugin	Opens a setup page for installing the site's plugin — it bundles the skill (and, when configured, the MCP server)	Claude Desktop / claude.ai / Cowork (Customize → Plugins → upload a custom plugin file) and Claude Code (terminal) — see below
Install Assistant	Installs the site's assistant app (PWA) so readers can launch the AI assistant from their home screen or desktop	Chromium browsers (others fall back to the standalone assistant page) — shown only when the assistant app is built

The generated artifacts (built once per `vark build`, no config beyond `markdownMenu`):

- `/_aardvark/<slug>-docs-skill.zip` — the downloadable skill (also published into the `/.well-known/agent-skills/index.json` discovery index above, so agents find it like your other skills).
- `/_aardvark/<slug>-docs-plugin.zip` — a single Claude plugin (the skill **and**, when `mcp: true` + a `baseUrl`, the MCP server) for the Claude apps: Customize → Plugins → upload a custom plugin file.
- `/_aardvark/<slug>-docs-marketplace.zip` — the same plugin wrapped in a local marketplace for the Claude Code terminal; add it with `/plugin marketplace add ./<slug>-docs-marketplace` then `/plugin install`.
- `/.well-known/claude-plugin/marketplace.json` — a hosted marketplace, so `/plugin marketplace add {baseUrl}/.well-known/claude-plugin/marketplace.json` installs the MCP server in one command (written when `installPlugin` is on — the default — plus `mcp: true` and a `baseUrl`).
- `/_aardvark/agent-setup.html` — the "Install Plugin" link target: a benefit-led page that walks a reader through installing the plugin in the **Claude apps** AND the **Claude Code** terminal.

To turn an item off, set it under `markdownMenu` (e.g. `markdownMenu: {copyMcp: false}`); every item defaults on. Copy MCP additionally requires `mcp: true`, and Install Assistant requires the assistant app to be built.

What only you can do

A static-site generator can't generate a cryptographic identity for you, operate your DNS, or run your identity provider. These steps are yours:

1. **Publish a Web Bot Auth key** (below) — required for the top Agent-Native level.
2. **Publish your DNS-AID records and enable DNSSEC** (below) — the one check no web server can pass for you.
3. (Optional, recommended) **Point OAuth/OIDC at your real identity provider** (below).
4. Set `baseUrl` and deploy over **HTTPS** on a real domain.

1. Publish a Web Bot Auth key

Web Bot Auth lets your site identify itself when an agent sends signed requests on its behalf. The scanner looks for a JWKS (a set of public keys) at `/.well-known/http-message-signatures-directory`. Aardvark **publishes** that directory — you just provide a public key.

Generate an Ed25519 keypair:

```
vark web-bot-auth-keygen

Private key — store as a secret; your signing agent needs it (NOT recoverable):
  <PRIVATE_KEY>

Public key — add to aardvark.config.yaml:

  webBotAuth:
    keys:
      - <PUBLIC_KEY>
```

Then add the **public** key to `aardvark.config.yaml`:

```
webBotAuth:
  keys:
    - <PUBLIC_KEY>   # the PUBLIC key from the command above
```

Listing a key turns the feature on; the next `vark build` writes the JWKS. Keep the **private** key out of your repo — store it as a secret wherever your signing agent runs (Aardvark never holds it, and the public directory is safe to commit). Aardvark only publishes the directory; it never signs requests itself. Full reference: Web Bot Auth.

2. Publish your DNS-AID records (and enable DNSSEC)

DNS-based Agent Interface Discovery is validated over DNS-over-HTTPS against your **authoritative nameservers** — so no HTTP server, `vark serve` included, can make it pass. Aardvark writes a copy-pasteable zone file for you at `/.well-known/dns-aid/records.zone`; you publish it:

1. `vark build` writes `records.zone` (and a `records.json` mirror) from your `dnsAid.services` config.
2. Paste those `SVCB`/`HTTPS` records into your DNS provider (Cloudflare DNS, Route 53, ...), matching the owner names and parameters exactly.
3. **Enable DNSSEC** for the zone so resolvers can authenticate the records.
4. Verify with a DoH lookup, e.g. `https://cloudflare-dns.com/dns-query?name=_mcp._agents.YOUR-DOMAIN&type=SVCB`.

Configuration and the full walkthrough live in Agent discovery.

3. Point OAuth at your IdP

Aardvark always advertises OAuth 2.1 / OpenID Connect discovery metadata (derived from `baseUrl` as labelled examples if you set nothing), so the discovery checks pass on a fresh build. For a real deployment, point the `oauth` block at your actual identity provider (for example Cloudflare Access) and fill in the `oauth.agentAuth` sub-block so `auth.md` advertises how an autonomous agent registers and proves identity:

```yaml
oauth:
  issuer: https://your-idp.example.com
  authorizationEndpoint: https://your-idp.example.com/authorize
  tokenEndpoint: https://your-idp.example.com/token
  jwksUri: https://your-idp.example.com/.well-known/jwks.json
  agentAuth:
    registerUri: https://your-idp.example.com/oauth/register
    identityTypesSupported: [service_account, delegated_user]
    credentialTypesSupported: [client_secret, private_key_jwt]
```

Aardvark is **not** an OAuth server — these documents only point agents at where to authenticate. See Agent discovery for every field.

Full checklist

Every scanner check, who handles it, and where to configure it:

Check	Category	Handled by	What to do
robots.txt	Discoverability	Aardvark	Nothing (tune `robots` if you like)
Sitemap	Discoverability	Aardvark	Set `baseUrl`

Link headers	Discoverability	Aardvark	Nothing
DNS-AID	Discoverability	**You**	Publish records + DNSSEC (step 2)
Markdown negotiation	Content accessibility	Aardvark	Nothing
robots.txt AI rules	Bot access control	Aardvark	Nothing
Content Signals	Bot access control	Aardvark	Tune `robots.contentSignals`
Web Bot Auth	Bot access control	**You**	Add a public key (step 1)
API catalog	Discovery	Aardvark	Embed `{% openapi %}` specs
OAuth/OIDC discovery	Discovery	Aardvark	Point at your IdP (step 3)
OAuth Protected Resource	Discovery	Aardvark	Nothing
auth.md	Discovery	Aardvark	Fill `oauth.agentAuth` for full credit
MCP Server Card	Discovery	Aardvark	Nothing
Agent Skills	Discovery	Aardvark	Add `skills/<name>/SKILL.md` (see Build-time AI)
WebMCP	Discovery	Aardvark	Enable `mcp: true` (opt-in)
A2A Agent Card	Discovery	Not yet	Aardvark does not emit an A2A agent card today
Commerce (x402, MPP, UCP, ACP, AP2)	Commerce	n/a	Not scored; only for commerce sites

Verify your score

After deploying, re-run the scan:

```
curl -X POST https://isitagentready.com/api/scan \
```

```
  -H 'Content-Type: application/json' \
  -d '{"url": "https://YOUR-SITE.com"}'
```

Or just open isitagentready.com and enter your URL. A check that reads `unableToCheck` is usually a fetch timeout, not a failure — re-run it. For Web Bot Auth specifically, confirm `checks.botAccessControl.webBotAuth.status` is `"pass"`.

Known gaps

- **A2A Agent Card** (`/.well-known/agent.json`): Aardvark advertises an A2A entry in its DNS-AID records but does not yet generate the agent card document itself. If you need this check, ship the card as a static file under `static/.well-known/`.
- **Commerce checks**: out of scope for a docs/marketing site and excluded from the score.

Agent discovery

Every `vark build` publishes a set of **always-on** discovery endpoints that let AI agents learn what your site offers and how to use it. These are emitted on every build (no opt-in needed), so a checker such as isitagentready.com clears with the best-available information. Each endpoint is populated from your config when you provide it, and otherwise from clearly-labelled **example** values derived from your `baseUrl`.

Endpoint	What it tells an agent
`/.well-known/mcp/server-card.json`	This site exposes an MCP server, where, and what it can do
`/.well-known/oauth-authorization-server`	OAuth 2.1 authorization-server metadata (RFC 8414)
`/.well-known/openid-configuration`	The OpenID Connect flavour of the same metadata
`/.well-known/oauth-protected-resource`	Which authorization server guards this resource (RFC 9728)
`/auth.md`	A human- and agent-readable summary of how to authenticate
`/.well-known/dns-aid/records.json` + `records.zone`	DNS records for agent-interface discovery

MCP Server Card

`/.well-known/mcp/server-card.json` advertises the Model Context Protocol server that `vark serve --mcp` hosts — its transport endpoint and the read-only documentation tools it exposes (`search_documentation`, `fetch_document`, `list_documentation`, `get_full_corpus`). The card is written on every build, so an agent can discover the server even on a static Cloudflare Pages / Netlify deploy where no live `/mcp` is running.

```
{
  "serverInfo": { "name": "Aardvark", "version": "0.1.8" },
  "protocolVersion": "2025-06-18",
  "endpoint": "https://aardvarkdocs.com/mcp",
  "transport": "streamable-http",
  "capabilities": { "tools": { "listChanged": false } },
  "tools": [ … ]
}
```

The endpoint is `{baseUrl}/mcp`. No configuration is required — the card reflects the tools the MCP server actually serves.

OAuth / OIDC discovery

Aardvark **publishes** OAuth 2.1 / OpenID Connect discovery metadata so an agent knows where to authenticate. It does **not** become an OAuth authorization server — these documents only advertise discovery information. Point them at your real identity provider (for example Cloudflare Access); left unset, every field is derived from `baseUrl` as a labelled example endpoint.

```
oauth:
  issuer: https://aardvarkdocs.com
  authorizationEndpoint: https://aardvarkdocs.com/oauth/authorize
  tokenEndpoint: https://aardvarkdocs.com/oauth/token
  jwksUri: https://aardvarkdocs.com/.well-known/jwks.json
  registrationEndpoint: https://aardvarkdocs.com/oauth/register
  scopesSupported: [openid, profile, email, offline_access]
  # Generic OAuth grants plus the two agent-auth profile grants (JWT-bearer assertion
exchange and
  # the WorkOS claim-ceremony polling grant); these are also the build default when the key
is unset.
  grantTypesSupported: [authorization_code, refresh_token, client_credentials,
urn:ietf:params:oauth:grant-type:jwt-bearer, urn:workos:agent-auth:grant-type:claim]
  responseTypesSupported: [code]
  oidc: true            # also publish /.well-known/openid-configuration (default on)
  agentAuth:            # how an autonomous agent registers and proves identity
    registerUri: https://aardvarkdocs.com/oauth/register
    claimUri: https://aardvarkdocs.com/oauth/claim
    revocationUri: https://aardvarkdocs.com/oauth/revoke
    identityTypesSupported: [identity_assertion, anonymous]
    assertionTypesSupported: [urn:ietf:params:oauth:token-type:id-jag, verified_email]
    credentialTypesSupported: [access_token]
    eventsSupported:
[https://schemas.workos.com/events/agent/auth/identity/assertion/revoked]
```

The build writes `/.well-known/oauth-authorization-server` (RFC 8414) and, when `oauth.oidc` is on (the default), `/.well-known/openid-configuration`. The discovery check accepts either, so a plain-OAuth site can set `oidc: false` to drop the OIDC copy. Both documents include an `agent_auth` block describing the agent registration / identity flow.

The `agent_auth` block

The `agent_auth` block follows the WorkOS auth.md profile's recognized registration flows. `identity_types_supported` lists the methods the service accepts, and each gets a nested object: `identity_assertion` carries the assertion types it accepts (the ID-JAG `urn:ietf:params:oauth:token-type:id-jag` and `verified_email`) plus the issued credential types, and `anonymous` carries its credential types. `events_supported` advertises the upstream revocation event the registration layer can ingest. Inside `agent_auth`, the registration / claim / revocation surface is published in BOTH vocabularies — the readiness scanner's `register_uri` / `claim_uri` / `revocation_uri` and the canonical WorkOS `identity_endpoint` / `claim_endpoint` / `events_endpoint` aliases — so either reader finds what it expects. Revocation is additionally exposed as the standard RFC 7009 `revocation_endpoint` at the **top level** of the authorization-server metadata (a sibling of `agent_auth`, not a field inside it):

```
{
  "revocation_endpoint": "https://aardvarkdocs.com/oauth/revoke",
  "agent_auth": {
    "skill": "https://aardvarkdocs.com/auth.md",
    "register_uri": "https://aardvarkdocs.com/oauth/register",
    "identity_endpoint": "https://aardvarkdocs.com/oauth/register",
    "claim_uri": "https://aardvarkdocs.com/oauth/claim",
    "claim_endpoint": "https://aardvarkdocs.com/oauth/claim",
    "revocation_uri": "https://aardvarkdocs.com/oauth/revoke",
    "events_endpoint": "https://aardvarkdocs.com/agent/event/notify",
    "identity_types_supported": ["identity_assertion", "anonymous"],
    "identity_assertion": {
      "assertion_types_supported": ["urn:ietf:params:oauth:token-type:id-jag",
"verified_email"],
      "credential_types_supported": ["access_token"]
    },
    "anonymous": { "credential_types_supported": ["access_token"] },
    "events_supported":
["https://schemas.workos.com/events/agent/auth/identity/assertion/revoked"]
  }
}
```

Protected Resource metadata

`/.well-known/oauth-protected-resource` (RFC 9728) names which authorization server guards the resource and how to present a token:

```
{
  "resource": "https://aardvarkdocs.com",
  "authorization_servers": ["https://aardvarkdocs.com"],
  "scopes_supported": ["openid", "profile", "email", "offline_access"],
  "bearer_methods_supported": ["header"]
}
```

auth.md

`/auth.md` restates the same facts in Markdown an agent (or a human) can read directly — its `# auth.md` heading is what readiness checkers look for. It also **embeds the authorization-server metadata, including the `agent_auth` block, as a fenced JSON code block** (mirroring the canonical WorkOS `AUTH.md`), so the `agent_auth` metadata is parseable from `auth.md` itself and not only from the separate well-known document. It is written on every build and served inline as `text/plain`.

DNS-AID records

DNS-based Agent Interface Discovery lets an agent find your machine interfaces from DNS alone, by looking up `SVCB`/`HTTPS` records under `_<service>._agents.<domain>`. Configure the services:

```
dnsAid:
  services:
    - name: a2a              # -> _a2a._agents.<domain>
      target: aardvarkdocs.com
      alpn: h2
      port: 443
      params:
        path: /.well-known/agent.json
    - name: mcp
      target: aardvarkdocs.com
      params: { path: /mcp }
```

`alpn` accepts either a single protocol (`alpn: h2`) or a list (`alpn: [h2, h3]`); a list is rendered as the comma-joined SVCB value `alpn="h2,h3"`.

Every build writes a copy-pasteable zone snippet at `/.well-known/dns-aid/records.zone` and a machine-readable mirror at `/.well-known/dns-aid/records.json`:

```
_a2a._agents.aardvarkdocs.com. 3600 IN SVCB 1 aardvarkdocs.com. alpn="h2" port=443
path="/.well-known/agent.json"
_mcp._agents.aardvarkdocs.com. 3600 IN SVCB 1 aardvarkdocs.com. alpn="h2" port=443
path="/mcp"
```

This step is manual (and required)

The DNS-AID check validates via **DNS-over-HTTPS against your domain's real authoritative nameservers** (`cloudflare-dns.com` / `dns.google`) — not against your web server. That means **no HTTP server, including `vark serve`, can make the DNS-AID check pass.** A static-site generator cannot operate your DNS or sign DNSSEC for you. To go green you must:

1. **Build** — `vark build` writes `/.well-known/dns-aid/records.zone` and `records.json`.
2. **Publish** the SVCB/HTTPS records from `records.zone` at your DNS provider (Cloudflare DNS, Route 53, etc.) — paste them into your zone, matching the owner names and SvcParams exactly.
3. **Enable DNSSEC** for the zone, so resolvers can authenticate the records.

4. **Verify** with a DoH lookup, e.g. `https://cloudflare-dns.com/dns-query?name=_mcp._agents.aardvarkdocs.com&type=SVCB`.

The `records.json` mirror served over HTTP is provided only for convenience and tooling — it cannot substitute for publishing the records at DNS.

How they're served

`/auth.md` and the `.json` artifacts are served by the normal static resolver. The extension-less OAuth/OIDC paths get dedicated `vark serve` routes (a suffix-less path is treated as an HTML request by the resolver, so the raw file would otherwise 404). On Cloudflare Pages / Netlify the generated `_headers` rules set `Content-Type: application/json` (with open CORS) on each. The homepage also advertises the MCP server card and the Protected Resource metadata via an RFC 8288 `Link` header.

Web Bot Auth

Web Bot Auth lets a site identify itself when an automated agent sends requests on its behalf. The agent signs each request with an Ed25519 key (HTTP Message Signatures, RFC 9421) and points a `Signature-Agent` header at a public **key directory**; the receiving site fetches that directory, looks up the key, and verifies the signature. Checkers such as isitagentready.com look for this directory at `/.well-known/http-message-signatures-directory`.

Aardvark **publishes** that directory. It never signs outbound requests itself, so it never holds a private key — you list your public key(s), and the matching private key lives wherever your signing agent runs.

Generate a key

`vark web-bot-auth-keygen` mints an Ed25519 keypair and prints both halves:

```
vark web-bot-auth-keygen

Generated an Ed25519 keypair for Web Bot Auth.

Private key — store as a secret; your signing agent needs it (NOT recoverable):
  <PRIVATE_KEY>

Public key — add to aardvark.config.yaml:

  webBotAuth:
    keys:
      - <PUBLIC_KEY>

Key ID (JWK thumbprint): <KEY_ID>
```

Store the **private** key as a secret for your agent — Aardvark never persists it, so save it now. Add the **public** key to your config.

Configure

```yaml
webBotAuth:
  keys:
    - <PUBLIC_KEY>   # a bare base64url Ed25519 public key
```

Each entry is either a bare public-key string (Aardvark fills in the JWK fields and computes the `kid` thumbprint) or a full JWK mapping if you want to pin `kid`, `alg`, or key-rotation windows (`nbf` / `exp`):

```yaml
webBotAuth:
  keys:
    - kty: OKP
      crv: Ed25519
      x: <PUBLIC_KEY>
      kid: <KEY_ID>     # pin the JWK thumbprint, optional
```

```
      alg: EdDSA        # pin the signature algorithm (EdDSA for an Ed25519 key), optional
      nbf: 1712793600   # not-before (Unix time), optional
      exp: 1715385600   # expires, optional — keep old + new keys overlapping when rotating
```

Listing `keys` turns the feature on. `webBotAuth: true` publishes an empty directory (valid, but the build warns until you add a key); `webBotAuth: { enabled: false }` turns it off even with keys present.

What gets published

`vark build` writes the directory as a JWKS to `/.well-known/http-message-signatures-directory`:

```
{
  "keys": [
    {
      "kty": "OKP",
      "crv": "Ed25519",
      "x": "<PUBLIC_KEY>",
      "kid": "<KEY_ID>",
      "use": "sig"
    }
  ]
}
```

It is served with `Content-Type: application/http-message-signatures-directory+json` and a one-day cache. The directory holds **public** key material only, so it is a plain static file: `vark serve` serves it directly, and on Cloudflare Pages / Netlify the generated `_headers` rule sets its media type. If you ship your own `static/.well-known/http-message-signatures-directory`, Aardvark leaves it untouched.

Plans & pricing

Aardvark is **free and open**: author, build, and self-host your docs at no cost, forever. Paid plans are for teams that want a **predictable monthly bill** — managed hosting, more seats, real support, and a monthly **included-AI allowance in dollars**, keeping pricing simple and dollar-based. Your Aardvark key also grants you access to hundreds of AI models you can use strategically to control costs as you go.

✨Free

$0 — pay as you go

Bring your own hosting, pay only for the AI you use.

- Self-host anywhere, full feature set
- All components, themes, agent discovery
- Metered AI at published per-model prices
- Prepaid balance + auto top-up
- 1 seat · community support

🚀Pro

$99 per month $87.20/mo billed annually

We host and run your docs.

- **Managed hosting** — custom domain, SSL, previews
- **$40/mo of AI included**, at a ≈7% member discount
- 5 seats included, add more at $10/seat
- Insights analytics
- Community support

🏢Business

$349 per month $303.20/mo billed annually

Robust and self-serve — no sales call.

- Everything in Pro, plus:
- **$120/mo of AI included** — 3× Pro — at a ≈13% discount
- 12 seats included, add more at $15/seat
- SSO, RBAC, audit-log export
- **Priority email support**, next-business-day target

Enterprise

$2,750 per month

Everything in Business, plus:

- **$750/mo of AI included** — 18.75× Pro — at a 20% discount
- **Unlimited customization requests** — our engineering team on call to build any feature you need
- SSO + SCIM provisioning
- Unlimited seats
- Shared Slack access for realtime support and requests, 7 days a week

Every model's real cost per answer, on every plan: **the pricing table**.

Compare plans

	Free	Pro	Business	Enterprise
Price	$0	$99/mo	$349/mo	$2,750/mo
Billed annually	—	$1,046.40/yr (≈$87.20/mo)	$3,638.40/yr (≈$303.20/mo)	$28,200/yr (≈$2,350/mo)
Included AI every month (member discount)	—	$40 at ≈7% off	$120 at ≈13% off (3× Pro)	$750 at 20% off (18.75× Pro)
Seats	1	5	12	Unlimited
Hosting	Self-host only	Managed or self-hosted	Managed or self-hosted	Managed or self-hosted
Analytics	Basic	Insights	Insights + export	Insights + export
SSO / RBAC / audit export	—	—	Included	+ SCIM
Customization	—	—	—	Unlimited requests
Support	Community	Community	Priority email	Realtime shared Slack
Self-hosting, agent discovery, PDF, all components	Included	Included	Included	Included

Nothing core is paywalled: self-hosting, Markdown-for-Agents, `llms.txt`, the agent-skills index, whole-site PDF, and every component and theme ship on **Free**. Paid plans add hosting, seats, support, controls — and cheaper AI.

The honest AI meter

Every competitor meters AI in credits you can't reason about. Aardvark bills AI in **dollars**: the pricing table publishes what every model costs per answer on every plan — real figures, no conversion math. Subscribers pay less per metered dollar the more they commit (≈7% off on Pro, ≈13% on Business, 20% on Enterprise), on top of the allowance their plan includes. All of it runs on **Aardvark's managed keys** — you never create, rotate, or secure a model-provider key on any plan.

Your included allowance is denominated in those same billed dollars:

- The dashboard shows **dollars included, dollars used**, your own trailing burn rate, and an estimated depletion date — no "≈N answers" marketing math.
- Unused allowance **rolls over one month**, up to 2× your monthly amount.
- Answers served by `:free` models never bill and never touch your allowance.

When the included AI runs out

Cap-and-hold is the default. Paid AI pauses at your cap and you're notified, so it cannot add further overflow charges. Your "max possible bill" is the subscription price plus any fixed add-on-seat charges and any overflow already incurred this month. (Usage-based GitHub Automations compute, if you use it, draws your prepaid balance separately and isn't included in that figure.) Prefer to stay unstuck? Two self-serve options, both under your control:

1. **Fall back to your balance** — flip overflow to draw from your prepaid balance at your plan's discounted member pricing, with a cap you set (raising it past 2× your allowance asks for an explicit acknowledgment).
2. **Auto top-up** — keep the balance funded automatically with your chosen amount and threshold, so overflow never stalls.

Annual billing

Annual plans take **20% off the platform fee** — the included-AI allowance is funded monthly at full value either way (it's real usage, not a discountable line item), so the effective saving is about 12–13% off the yearly total — ~12% on Pro, ~13% on Business (the bigger a plan's fee is relative to its bundled allowance, the more of the bill the discount reaches). Pro: $1,046.40/yr (vs. $1,188 at par, ~12% off). Business: $3,638.40/yr (vs. $4,188, ~13% off). The allowance still resets every month on annual plans.

FAQ

Billing is designed to be predictable and never to surprise you. Here's exactly what happens in the situations people ask about most.

Every question — and its exact answer — now lives in the searchable support knowledge base, organized by category with the most-asked questions up top.

🛟Browse the support knowledge base

Included AI, overflow, plan changes, payments and safety limits, seats and hosting — every billing question answered, straight from the team.

AI model rates

Per-token rates for every model Aardvark's metered gateway can serve — point `ai.model` / `ai.assistant.model` at **any** model below and pay only for the AI you actually use. Looking for the platform tiers (hosting, seats, support, included AI)? See **Plans & pricing**.

Input/**Output** are the pay-as-you-go per-token prices; the per-plan columns show what one assistant answer actually costs on each tier. Subscribers get a member discount on every metered dollar — about **7% off on Pro**, **13% off on Business**, and **20% off on Enterprise** — on top of their plan's included monthly AI allowance.

Reading the columns

- **Input** / **Output** are the pay-as-you-go prices per **one million tokens**.
- The **est./answer** columns estimate one assistant turn on each plan — roughly 68K tokens of inlined docs context plus a ~600-token reply (capped at each model's context window). Input dominates, so a single answer costs far less than the per-million output rate suggests; your real numbers vary with your docs size and answer length. On a paid plan, answers drain the plan's included monthly allowance before anything is billed.
- **Quality** is the independent Artificial Analysis Intelligence Index (higher is better); it's blank for models Artificial Analysis doesn't cover.

Recommended (auto-updating)

These floating aliases always route to the provider's newest model, so you never pin a version. Aardvark's default config uses `~anthropic/claude-sonnet-latest`.

Model	Slug	Type	Context	Quality	Input	Output	Free est./answer	Pro est./answer	Business est./answer	Enterprise est./answer
OpenAI: GPT Chat Latest	`openai/gpt-chat-latest`	Text + file/image	400K	60.9	$7.50	$45.00	$0.54	$0.50	$0.47	$0.43
Anthropic: Claude Fable Latest	`~anthropic/claude-fable-latest`	Text + file/image	1M	62.1	$15.00	$75.00	$1.07	$0.99	$0.92	$0.85
Anthropic Claude Haiku Latest	`~anthropic/claude-haiku-latest`	Text + file/image	200K	29.9	$1.50	$7.50	$0.11	$0.10	$0.09	$0.09
Anthropic: Claude Opus Latest	`~anthropic/claude-opus-latest`	Text + file/image	1M	63.1	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic Claude Sonnet Latest	`~anthropic/claude-sonnet-latest`	Text + file/image	1M	55.3	$3.00	$15.00	$0.21	$0.20	$0.18	$0.17
DeepSeek V4 Flash Latest	`~deepseek/deepseek-v4-flash-latest`	Text	1.3M	51.8	$0.11	$0.23	$0.01	$0.01	$0.01	$0.01
Google Gemini Flash Latest	`~google/gemini-flash-latest`	Text + audio/file/image/video	1.0M	56.0	$0.56	$2.81	$0.04	$0.04	$0.03	$0.03

Google Gemini Pro Latest	`~google/gemini-pro-latest`	Text + audio/file/image/video	1.0M	47.7	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
MoonshotAI Kimi Latest	`~moonshotai/kimi-latest`	Text + image/video	1.0M	59.7	$3.90	$19.50	$0.28	$0.26	$0.24	$0.22
OpenAI GPT Latest	`~openai/gpt-latest`	Text + file/image	1.1M	60.9	$3.75	$22.50	$0.27	$0.25	$0.23	$0.21
OpenAI GPT Mini Latest	`~openai/gpt-mini-latest`	Text + file/image	400K	40.9	$1.12	$6.75	$0.08	$0.08	$0.07	$0.06
xAI: Grok Latest	`~x-ai/grok-latest`	Text + file/image	500K	7.3	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
Z.ai: GLM Latest	`~z-ai/glm-latest`	Text	1.0M	59.5	$2.10	$6.60	$0.15	$0.14	$0.13	$0.12

All models

Free ($0.00) models come with hard limits

A model priced **$0.00** runs on a shared upstream **free pool** — handy for trying things out, but with real trade-offs you should plan around:

- **Slow and queue-bound.** Free requests are low priority and share a queue, so responses are slower and degrade further under load.
- **Capped per day and per minute.** Free usage is limited to roughly a few dozen requests per day plus a low per-minute rate; once you hit the cap, requests are refused until the window resets.
- **Best-effort availability.** Free endpoints can be throttled, deprecated, or briefly unavailable with no notice.
- **Your inputs may train models.** Some free providers may use prompts for training, so don't send anything sensitive through a free model.

For a public **Ask AI assistant** under real traffic, choose a paid model — the limits above will otherwise rate-limit or stall your readers.

Click a column header to sort. 411 models with published per-token pricing are available.

Model	Slug	Type	Context	Quality	Input	Output	Free est./answer	Pro est./answer	Business est./answer	Enterprise est./answer
AI21: Jamba Large 1.7	`ai21/jamba-large-1.7`	Text	256K	5.0	$3.00	$12.00	$0.21	$0.20	$0.18	$0.17
AionLabs: Aion-2.0	`aion-labs/aion-2.0`	Text	131K	—	$1.20	$2.40	$0.08	$0.08	$0.07	$0.07
AionLabs: Aion-3.0	`aion-labs/aion-3.0`	Text	131K	—	$4.50	$9.00	$0.31	$0.29	$0.27	$0.25
AionLabs: Aion-3.0-Mini	`aion-labs/aion-3.0-mini`	Text	131K	—	$1.05	$2.10	$0.07	$0.07	$0.06	$0.06
AionLabs: Aion-RP 1.0 (8B)	`aion-labs/aion-rp-llama-3.1-8b`	Text	32K	—	$1.20	$2.40	$0.04	$0.04	$0.03	$0.03
AllenAI: Olmo 3 32B Think	`allenai/olmo-3-32b-think`	Text	65K	6.1	$0.22	$0.75	$0.02	$0.01	$0.01	$0.01
Amazon: Nova 2 Lite	`amazon/nova-2-lite-v1`	Text + file/image/video	1M	—	$0.45	$3.75	$0.03	$0.03	$0.03	$0.03
Amazon: Nova Lite 1.0	`amazon/nova-lite-v1`	Text + image	300K	—	$0.09	$0.36	$0.01	$0.01	$0.01	$0.01
Amazon: Nova Micro 1.0	`amazon/nova-micro-v1`	Text	128K	—	$0.05	$0.21	$0.00	$0.00	$0.00	$0.00
Amazon: Nova Premier 1.0	`amazon/nova-premier-v1`	Text + image	1M	—	$3.75	$18.75	$0.27	$0.25	$0.23	$0.21
Amazon: Nova Pro 1.0	`amazon/nova-pro-v1`	Text + image	300K	—	$1.20	$4.80	$0.08	$0.08	$0.07	$0.07
Magnum v4 72B	`anthracite-org/magnum-v4-72b`	Text	32K	—	$4.50	$7.50	$0.15	$0.14	$0.13	$0.12

Anthropic: Claude 3 Haiku	anthropic/claude-3-haiku	Text + image	200K	3.5	$0.38	$1.88	$0.03	$0.02	$0.02	$0.02
Anthropic: Claude Fable 5	anthropic/claude-fable-5	Text + file/image	1M	62.1	$15.00	$75.00	$1.07	$0.99	$0.92	$0.85
Anthropic: Claude Fable 5 (batch)	anthropic/claude-fable-5:batch	Text + file/image	1M	62.1	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic: Claude Haiku 4.5	anthropic/claude-haiku-4.5	Text + file/image	200K	24.1	$1.50	$7.50	$0.11	$0.10	$0.09	$0.09
Anthropic: Claude Haiku 4.5 (batch)	anthropic/claude-haiku-4.5:batch	Text + file/image	200K	24.1	$0.75	$3.75	$0.05	$0.05	$0.05	$0.04
Anthropic: Claude Opus 4	anthropic/claude-opus-4	Text + file/image	200K	26.0	$22.50	$112.50	$1.60	$1.49	$1.38	$1.28
Anthropic: Claude Opus 4.1	anthropic/claude-opus-4.1	Text + file/image	200K	28.8	$22.50	$112.50	$1.60	$1.49	$1.38	$1.28
Anthropic: Claude Opus 4.1 (batch)	anthropic/claude-opus-4.1:batch	Text + file/image	200K	28.8	$11.25	$56.25	$0.80	$0.75	$0.69	$0.64
Anthropic: Claude Opus 4.5	anthropic/claude-opus-4.5	Text + file/image	200K	35.6	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic: Claude Opus 4.5 (batch)	anthropic/claude-opus-4.5:batch	Text + file/image	200K	35.6	$3.75	$18.75	$0.27	$0.25	$0.23	$0.21
Anthropic: Claude Opus 4.6	anthropic/claude-opus-4.6	Text + file/image	1M	38.8	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic: Claude Opus 4.6 (batch)	anthropic/claude-opus-4.6:batch	Text + file/image	1M	38.8	$3.75	$18.75	$0.27	$0.25	$0.23	$0.21
Anthropic: Claude Opus 4.7	anthropic/claude-opus-4.7	Text + file/image	1M	55.0	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic: Claude Opus 4.7 (Fast)	anthropic/claude-opus-4.7-fast	Text + file/image	1M	55.0	$45.00	$225.00	$3.19	$2.98	$2.77	$2.56
Anthropic: Claude Opus 4.7 (batch)	anthropic/claude-opus-4.7:batch	Text + file/image	1M	55.0	$3.75	$18.75	$0.27	$0.25	$0.23	$0.21
Anthropic: Claude Opus 4.8	anthropic/claude-opus-4.8	Text + file/image	1M	57.3	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic: Claude Opus 4.8 (Fast)	anthropic/claude-opus-4.8-fast	Text + file/image	1M	57.3	$15.00	$75.00	$1.07	$0.99	$0.92	$0.85
Anthropic: Claude Opus 4.8 (batch)	anthropic/claude-opus-4.8:batch	Text + file/image	1M	57.3	$3.75	$18.75	$0.27	$0.25	$0.23	$0.21
Claude Opus 5	anthropic/claude-opus-5	Text + file/image	1M	63.1	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Claude Opus 5 (Fast)	anthropic/claude-opus-5-fast	Text + file/image	1M	63.1	$15.00	$75.00	$1.07	$0.99	$0.92	$0.85
Claude Opus 5 (batch)	anthropic/claude-opus-5:batch	Text + file/image	1M	63.1	$3.75	$18.75	$0.27	$0.25	$0.23	$0.21
Anthropic: Claude Sonnet 4	anthropic/claude-sonnet-4	Text + file/image	1M	26.0	$4.50	$22.50	$0.32	$0.30	$0.28	$0.26
Anthropic: Claude Sonnet 4.5	anthropic/claude-sonnet-4.5	Text + file/image	1M	29.9	$4.50	$22.50	$0.32	$0.30	$0.28	$0.26
Anthropic: Claude Sonnet 4.5 (batch)	anthropic/claude-sonnet-4.5:batch	Text + file/image	1M	29.9	$2.25	$11.25	$0.16	$0.15	$0.14	$0.13
Anthropic: Claude Sonnet 4.6	anthropic/claude-sonnet-4.6	Text + file/image	1M	36.8	$4.50	$22.50	$0.32	$0.30	$0.28	$0.26
Anthropic: Claude Sonnet 4.6 (batch)	anthropic/claude-sonnet-4.6:batch	Text + file/image	1M	36.8	$2.25	$11.25	$0.16	$0.15	$0.14	$0.13
Anthropic: Claude Sonnet 5	anthropic/claude-sonnet-5	Text + file/image	1M	55.3	$3.00	$15.00	$0.21	$0.20	$0.18	$0.17
Anthropic: Claude Sonnet 5 (batch)	anthropic/claude-sonnet-5:batch	Text + file/image	1M	55.3	$1.50	$7.50	$0.11	$0.10	$0.09	$0.09
Arcee AI: Trinity Large Thinking	arcee-ai/trinity-large-thinking	Text	262K	18.7	$0.33	$1.27	$0.02	$0.02	$0.02	$0.02
Arcee AI: Virtuoso Large	arcee-ai/virtuoso-large	Text	131K	—	$1.12	$1.80	$0.08	$0.07	$0.07	$0.06
Baidu: ERNIE 4.5 VL 424B A47B	baidu/ernie-4.5-vl-424b-a47b	Text + image	123K	—	$0.63	$1.88	$0.04	$0.04	$0.04	$0.04
ByteDance Seed: Seed 1.6	bytedance-seed/seed-1.6	Text + image/video	262K	—	$0.38	$3.00	$0.03	$0.03	$0.02	$0.02
ByteDance Seed: Seed 1.6 Flash	bytedance-seed/seed-1.6-flash	Text + image/video	262K	—	$0.11	$0.45	$0.01	$0.01	$0.01	$0.01
ByteDance Seed: Seed 2.1 Turbo	bytedance-seed/seed-2-1-turbo	Text + image/video	262K	—	$0.75	$3.75	$0.05	$0.05	$0.05	$0.04
ByteDance Seed: Seed-2.0-Code	bytedance-seed/seed-2.0-code	Text + image/video	262K	—	$0.75	$4.50	$0.05	$0.05	$0.05	$0.04

ByteDance Seed: Seed-2.0-Lite	`bytedance-seed/seed-2.0-lite`	Text + image/video	262K	—	$0.38	$3.00	$0.03	$0.03	$0.02	$0.02
ByteDance Seed: Seed-2.0-Mini	`bytedance-seed/seed-2.0-mini`	Text + image/video	262K	—	$0.15	$0.60	$0.01	$0.01	$0.01	$0.01
ByteDance: UI-TARS 7B	`bytedance/ui-tars-1.5-7b`	Text + image	128K	—	$0.15	$0.30	$0.01	$0.01	$0.01	$0.01
Venice: Uncensored	`cognitivecomputations/dolphin-mistral-24b-venice-edition`	Text	128K	—	$0.30	$1.35	$0.02	$0.02	$0.02	$0.02
Cohere: Command A	`cohere/command-a`	Text	256K	22.8	$3.75	$15.00	$0.26	$0.25	$0.23	$0.21
Cohere: Command R (08-2024)	`cohere/command-r-08-2024`	Text	128K	—	$0.22	$0.90	$0.02	$0.01	$0.01	$0.01
Cohere: Command R+ (08-2024)	`cohere/command-r-plus-08-2024`	Text	128K	—	$3.75	$15.00	$0.26	$0.25	$0.23	$0.21
Cohere: Command R7B (12-2024)	`cohere/command-r7b-12-2024`	Text	128K	—	$0.06	$0.22	$0.00	$0.00	$0.00	$0.00
Cohere: North Mini Code (free)	`cohere/north-mini-code:free`	Text	256K	20.2	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Deep Cogito: Cogito v2.1 671B	`deepcogito/cogito-v2.1-671b`	Text	128K	—	$1.88	$1.88	$0.13	$0.12	$0.11	$0.10
DeepSeek: DeepSeek V3	`deepseek/deepseek-chat`	Text	163K	14.2	$0.39	$1.54	$0.03	$0.03	$0.02	$0.02
DeepSeek: DeepSeek V3 0324	`deepseek/deepseek-chat-v3-0324`	Text	163K	15.2	$0.38	$1.50	$0.03	$0.02	$0.02	$0.02
DeepSeek: DeepSeek V3.1	`deepseek/deepseek-chat-v3.1`	Text	163K	21.4	$0.38	$1.42	$0.03	$0.02	$0.02	$0.02
DeepSeek: R1	`deepseek/deepseek-r1`	Text	64K	20.4	$1.05	$3.75	$0.07	$0.06	$0.06	$0.06
DeepSeek: R1 0528	`deepseek/deepseek-r1-0528`	Text	163K	6.0	$0.75	$3.23	$0.05	$0.05	$0.05	$0.04
DeepSeek: R1 Distill Llama 70B	`deepseek/deepseek-r1-distill-llama-70b`	Text	8K	9.8	$1.20	$1.20	$0.01	$0.01	$0.01	$0.01
DeepSeek: DeepSeek V3.1 Terminus	`deepseek/deepseek-v3.1-terminus`	Text	163K	21.7	$0.41	$1.50	$0.03	$0.03	$0.02	$0.02
DeepSeek: DeepSeek V3.2	`deepseek/deepseek-v3.2`	Text	163K	25.1	$0.40	$0.60	$0.03	$0.03	$0.02	$0.02
DeepSeek: DeepSeek V3.2 Exp	`deepseek/deepseek-v3.2-exp`	Text	163K	32.8	$0.41	$0.61	$0.03	$0.03	$0.02	$0.02
DeepSeek: DeepSeek V4 Flash 0423	`deepseek/deepseek-v4-flash`	Text	1.0M	51.8	$0.12	$0.25	$0.01	$0.01	$0.01	$0.01
DeepSeek: DeepSeek V4 Flash 0731	`deepseek/deepseek-v4-flash-0731`	Text	1.3M	51.8	$0.21	$0.42	$0.01	$0.01	$0.01	$0.01
DeepSeek: DeepSeek V4 Pro 0423	`deepseek/deepseek-v4-pro`	Text	1.0M	53.2	$2.16	$4.32	$0.15	$0.14	$0.13	$0.12
DeepSeek: DeepSeek V4 Pro 0813	`deepseek/deepseek-v4-pro-0813`	Text	1.0M	53.2	$0.99	$2.97	$0.07	$0.06	$0.06	$0.06
Dots Studio: Dots3-Note Preview (free)	`dots-studio/dots-3-note-preview:free`	Text + image	512K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Google: Gemini 2.5 Flash	`google/gemini-2.5-flash`	Text + audio/file/image/video	1.0M	14.2	$0.45	$3.75	$0.03	$0.03	$0.03	$0.03
Google: Nano Banana (Gemini 2.5 Flash Image)	`google/gemini-2.5-flash-image`	Multimodal out	32K	—	$0.45	$3.75	$0.02	$0.02	$0.01	$0.01
Google: Gemini 2.5 Flash Lite	`google/gemini-2.5-flash-lite`	Text + audio/file/image/video	1.0M	6.7	$0.15	$0.60	$0.01	$0.01	$0.01	$0.01
Google: Gemini 2.5 Flash Lite (batch)	`google/gemini-2.5-flash-lite:batch`	Text + audio/file/image/video	1.0M	6.7	$0.07	$0.30	$0.01	$0.00	$0.00	$0.00
Google: Gemini 2.5 Flash (batch)	`google/gemini-2.5-flash:batch`	Text + audio/file/image/video	1.0M	14.2	$0.22	$1.88	$0.02	$0.02	$0.01	$0.01
Google: Gemini 2.5 Pro	`google/gemini-2.5-pro`	Text + audio/file/image/video	1.0M	25.9	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
Google: Gemini 2.5 Pro Preview 06-05	`google/gemini-2.5-pro-preview`	Text + audio/file/image	1.0M	25.9	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
Google: Gemini 2.5 Pro Preview 05-06	`google/gemini-2.5-pro-preview-05-06`	Text + audio/file/image/video	1.0M	22.7	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
Google: Gemini 2.5 Pro (batch)	`google/gemini-2.5-pro:batch`	Text + audio/file/image/video	1.0M	25.9	$0.94	$7.50	$0.07	$0.06	$0.06	$0.05
Google: Gemini 3 Flash Preview	`google/gemini-3-flash-preview`	Text + audio/file/image/video	1.0M	38.7	$0.75	$4.50	$0.05	$0.05	$0.05	$0.04
Google: Gemini 3 Flash Preview (batch)	`google/gemini-3-flash-preview:batch`	Text + audio/file/image/video	1.0M	38.7	$0.38	$2.25	$0.03	$0.03	$0.02	$0.02

Google: Nano Banana Pro (Gemini 3 Pro Image)	google/gemini-3-pro-image	Multimodal out	131K	—	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
Google: Nano Banana Pro (Gemini 3 Pro Image Preview)	google/gemini-3-pro-image-preview	Multimodal out	65K	—	$3.00	$18.00	$0.21	$0.19	$0.18	$0.16
Google: Nano Banana 2 (Gemini 3.1 Flash Image)	google/gemini-3.1-flash-image	Multimodal out	131K	—	$0.75	$4.50	$0.05	$0.05	$0.05	$0.04
Google: Nano Banana 2 (Gemini 3.1 Flash Image Preview)	google/gemini-3.1-flash-image-preview	Multimodal out	65K	—	$0.75	$4.50	$0.05	$0.05	$0.04	$0.04
Google: Gemini 3.1 Flash Lite	google/gemini-3.1-flash-lite	Text + audio/file/image/video	1.0M	25.6	$0.38	$2.25	$0.03	$0.03	$0.02	$0.02
Google: Nano Banana 2 Lite (Gemini 3.1 Flash Lite Image)	google/gemini-3.1-flash-lite-image	Multimodal out	65K	—	$0.38	$2.25	$0.03	$0.02	$0.02	$0.02
Google: Gemini 3.1 Flash Lite Preview	google/gemini-3.1-flash-lite-preview	Text + audio/file/image/video	1.0M	25.6	$0.38	$2.25	$0.03	$0.03	$0.02	$0.02
Google: Gemini 3.1 Flash Lite (batch)	google/gemini-3.1-flash-lite:batch	Text + audio/file/image/video	1.0M	25.6	$0.19	$1.12	$0.01	$0.01	$0.01	$0.01
Google: Gemini 3.1 Pro Preview	google/gemini-3.1-pro-preview	Text + audio/file/image/video	1.0M	47.7	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
Google: Gemini 3.1 Pro Preview Custom Tools	google/gemini-3.1-pro-preview-customtools	Text + audio/file/image/video	1.0M	—	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
Google: Gemini 3.1 Pro Preview (batch)	google/gemini-3.1-pro-preview:batch	Text + audio/file/image/video	1.0M	47.7	$1.50	$9.00	$0.11	$0.10	$0.09	$0.09
Google: Gemini 3.5 Flash	google/gemini-3.5-flash	Text + audio/file/image/video	1.0M	52.0	$2.25	$13.50	$0.16	$0.15	$0.14	$0.13
Google: Gemini 3.5 Flash Lite	google/gemini-3.5-flash-lite	Text + audio/file/image/video	1.0M	37.4	$0.45	$3.75	$0.03	$0.03	$0.03	$0.03
Google: Gemini 3.5 Flash Lite (batch)	google/gemini-3.5-flash-lite:batch	Text + audio/file/image/video	1.0M	37.4	$0.22	$1.88	$0.02	$0.02	$0.01	$0.01
Google: Gemini 3.5 Flash (batch)	google/gemini-3.5-flash:batch	Text + audio/file/image/video	1.0M	52.0	$1.12	$6.75	$0.08	$0.08	$0.07	$0.06
Google: Gemini 3.6 Flash	google/gemini-3.6-flash	Text + audio/file/image/video	1.0M	51.6	$1.12	$5.62	$0.08	$0.07	$0.07	$0.06
Google: Gemini 3.6 Flash (batch)	google/gemini-3.6-flash:batch	Text + audio/file/image/video	1.0M	51.6	$0.56	$2.81	$0.04	$0.04	$0.03	$0.03
Google: Gemini 3.7 Flash	google/gemini-3.7-flash	Text + audio/file/image/video	1.0M	56.0	$0.56	$2.81	$0.04	$0.04	$0.03	$0.03
Google: Gemini 3.7 Flash (batch)	google/gemini-3.7-flash:batch	Text + audio/file/image/video	1.0M	56.0	$0.28	$1.41	$0.02	$0.02	$0.02	$0.02
Google: Gemma 2 27B	google/gemma-2-27b-it	Text	8K	—	$0.98	$0.98	$0.01	$0.01	$0.01	$0.01
Google: Gemma 3 12B	google/gemma-3-12b-it	Text + image	131K	5.5	$0.07	$0.22	$0.01	$0.00	$0.00	$0.00
Google: Gemma 3 27B	google/gemma-3-27b-it	Text + image	262K	7.4	$0.12	$0.67	$0.01	$0.01	$0.01	$0.01
Google: Gemma 3 4B	google/gemma-3-4b-it	Text + image	131K	1.0	$0.07	$0.15	$0.01	$0.00	$0.00	$0.00
Google: Gemma 3n 4B	google/gemma-3n-e4b-it	Text	32K	—	$0.09	$0.18	$0.00	$0.00	$0.00	$0.00
Google: Gemma 4 26B A4B	google/gemma-4-26b-a4b-it	Text + image/video	262K	26.1	$0.11	$0.51	$0.01	$0.01	$0.01	$0.01
Google: Gemma 4 26B A4B (free)	google/gemma-4-26b-a4b-it:free	Text + image/video	262K	26.1	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Google: Gemma 4 31B	google/gemma-4-31b-it	Text + image/video	262K	29.7	$0.14	$0.51	$0.01	$0.01	$0.01	$0.01
Google: Gemma 4 31B (free)	google/gemma-4-31b-it:free	Text + image/video	262K	29.7	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Google: Lyria 3 Clip Preview	google/lyria-3-clip-preview	Multimodal out	1.0M	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Google: Lyria 3 Pro Preview	google/lyria-3-pro-preview	Multimodal out	1.0M	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
MythoMax 13B	gryphe/mythomax-l2-13b	Text	8K	—	$0.09	$0.09	$0.00	$0.00	$0.00	$0.00
IBM: Granite 4.0 Micro	ibm-granite/granite-4.0-h-micro	Text	131K	2.0	$0.03	$0.17	$0.00	$0.00	$0.00	$0.00
IBM: Granite 4.1 8B	ibm-granite/granite-4.1-8b	Text	131K	6.4	$0.07	$0.15	$0.01	$0.00	$0.00	$0.00
Inception: Mercury 2	inception/mercury-2	Text	128K	21.9	$0.38	$1.12	$0.03	$0.02	$0.02	$0.02
inclusionAI: Ling-2.6-1T	inclusionai/ling-2.6-1t	Text	262K	26.6	$0.11	$0.94	$0.01	$0.01	$0.01	$0.01
inclusionAI: Ling-2.6-flash	inclusionai/ling-2.6-flash	Text	262K	14.2	$0.01	$0.04	$0.00	$0.00	$0.00	$0.00

Ling-3.0-flash	inclusionai/ling-3.0-flash	Text	262K	37.8	$0.03	$0.09	$0.00	$0.00	$0.00	$0.00
inclusionAI: Ring-2.6-1T	inclusionai/ring-2.6-1t	Text	262K	31.7	$0.11	$0.94	$0.01	$0.01	$0.01	$0.01
Kwaipilot: KAT-Coder-Air V2.5	kwaipilot/kat-coder-air-v2.5	Text	256K	—	$0.22	$0.90	$0.02	$0.01	$0.01	$0.01
Kwaipilot: KAT-Coder-Pro V2	kwaipilot/kat-coder-pro-v2	Text	262K	33.7	$0.45	$1.80	$0.03	$0.03	$0.03	$0.03
Kwaipilot: KAT-Coder-Pro V2.5	kwaipilot/kat-coder-pro-v2.5	Text	256K	—	$1.11	$4.44	$0.08	$0.07	$0.07	$0.06
LiquidAI: LFM2.5-2.6B (free)	liquid/lfm-2.5-2.6b:free	Text	128K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Mancer: Weaver (alpha)	mancer/weaver	Text	8K	—	$0.75	$1.12	$0.01	$0.01	$0.01	$0.00
Meituan: LongCat 2.0	meituan/longcat-2.0	Text	1.0M	34.0	$0.45	$1.80	$0.03	$0.03	$0.03	$0.03
Meta: Llama 3.1 70B Instruct	meta-llama/llama-3.1-70b-instruct	Text	131K	6.5	$0.60	$0.60	$0.04	$0.04	$0.04	$0.03
Meta: Llama 3.1 8B Instruct	meta-llama/llama-3.1-8b-instruct	Text	131K	7.4	$0.07	$0.12	$0.01	$0.00	$0.00	$0.00
Meta: Llama 3.2 1B Instruct	meta-llama/llama-3.2-1b-instruct	Text	60K	1.0	$0.04	$0.30	$0.00	$0.00	$0.00	$0.00
Meta: Llama 3.2 3B Instruct	meta-llama/llama-3.2-3b-instruct	Text	131K	3.9	$0.07	$0.49	$0.01	$0.01	$0.00	$0.00
Meta: Llama 3.3 70B Instruct	meta-llama/llama-3.3-70b-instruct	Text	131K	9.3	$0.15	$0.48	$0.01	$0.01	$0.01	$0.01
Meta: Llama 4 Maverick	meta-llama/llama-4-maverick	Text + image	1.0M	14.5	$0.30	$1.20	$0.02	$0.02	$0.02	$0.02
Meta: Llama 4 Scout	meta-llama/llama-4-scout	Text + image	1.3M	10.3	$0.15	$0.45	$0.01	$0.01	$0.01	$0.01
Meta: Llama Guard 4 12B	meta-llama/llama-guard-4-12b	Text + image	1.0M	—	$0.27	$0.27	$0.02	$0.02	$0.02	$0.01
Meta: Muse Glimmer 30B	meta/muse-glimmer-30b	Text + image	131K	—	$0.52	$2.25	$0.04	$0.03	$0.03	$0.03
Meta: Muse Spark 1.1	meta/muse-spark-1.1	Text + audio/file/image/video	1.0M	53.2	$1.88	$6.38	$0.13	$0.12	$0.11	$0.11
Meta: Muse Spark 1.2	meta/muse-spark-1.2	Text + audio/file/image/video	1.0M	56.8	$1.88	$6.38	$0.13	$0.12	$0.11	$0.11
Microsoft: Phi 4	microsoft/phi-4	Text	16K	4.6	$0.11	$0.21	$0.00	$0.00	$0.00	$0.00
WizardLM-2 8x22B	microsoft/wizardlm-2-8x22b	Text	65K	—	$0.93	$0.93	$0.06	$0.06	$0.05	$0.05
MiniMax: MiniMax-01	minimax/minimax-01	Text + image	1.0M	—	$0.30	$1.65	$0.02	$0.02	$0.02	$0.02
MiniMax: MiniMax M1	minimax/minimax-m1	Text	1M	—	$0.83	$3.30	$0.06	$0.05	$0.05	$0.05
MiniMax: MiniMax M2	minimax/minimax-m2	Text	204K	28.9	$0.38	$1.53	$0.03	$0.03	$0.02	$0.02
MiniMax: MiniMax M2-her	minimax/minimax-m2-her	Text	65K	—	$0.45	$1.80	$0.03	$0.03	$0.03	$0.02
MiniMax: MiniMax M2.1	minimax/minimax-m2.1	Text	204K	32.1	$0.45	$1.80	$0.03	$0.03	$0.03	$0.03
MiniMax: MiniMax M2.5	minimax/minimax-m2.5	Text	204K	34.5	$0.34	$1.35	$0.02	$0.02	$0.02	$0.02
MiniMax: MiniMax M2.7	minimax/minimax-m2.7	Text	204K	38.9	$0.45	$1.80	$0.03	$0.03	$0.03	$0.03
MiniMax: MiniMax M3	minimax/minimax-m3	Text + image/video	1.0M	45.4	$0.45	$1.80	$0.03	$0.03	$0.03	$0.03
MiniMax: MiniMax M3 (batch)	minimax/minimax-m3:batch	Text + image/video	524K	45.4	$0.45	$1.80	$0.03	$0.03	$0.03	$0.03
Mistral: Codestral 2508	mistralai/codestral-2508	Text + file	256K	—	$0.45	$1.35	$0.03	$0.03	$0.03	$0.03
Mistral: Ministral 3 14B 2512	mistralai/ministral-14b-2512	Text + image	262K	11.2	$0.30	$0.30	$0.02	$0.02	$0.02	$0.02
Mistral: Ministral 3 3B 2512	mistralai/ministral-3b-2512	Text + image	131K	7.1	$0.15	$0.15	$0.01	$0.01	$0.01	$0.01
Mistral: Ministral 3 8B 2512	mistralai/ministral-8b-2512	Text + image	262K	9.0	$0.22	$0.22	$0.02	$0.01	$0.01	$0.01
Mistral Large	mistralai/mistral-large	Text + file	128K	4.1	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
Mistral Large 2407	mistralai/mistral-large-2407	Text + file	131K	7.0	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
Mistral: Mistral Large 3 2512	mistralai/mistral-large-2512	Text + file/image	262K	15.9	$0.75	$2.25	$0.05	$0.05	$0.05	$0.04
Mistral: Mistral Medium 3	mistralai/mistral-medium-3	Text + file/image	131K	12.5	$0.60	$3.00	$0.04	$0.04	$0.04	$0.03
Mistral: Mistral Medium 3.5	mistralai/mistral-medium-3-5	Text + file/image	262K	30.4	$2.25	$11.25	$0.16	$0.15	$0.14	$0.13

Mistral: Mistral Medium 3.1	mistralai/mistral-medium-3.1	Text + file/image	131K	14.7	$0.60	$3.00	$0.04	$0.04	$0.04	$0.03
Mistral: Mistral Nemo	mistralai/mistral-nemo	Text	131K	—	$0.03	$0.04	$0.00	$0.00	$0.00	$0.00
Mistral: Saba	mistralai/mistral-saba	Text + file	32K	6.2	$0.30	$0.90	$0.01	$0.01	$0.01	$0.01
Mistral: Mistral Small 3	mistralai/mistral-small-24b-instruct-2501	Text	32K	6.7	$0.07	$0.12	$0.00	$0.00	$0.00	$0.00
Mistral: Mistral Small 4	mistralai/mistral-small-2603	Text + image	262K	19.7	$0.22	$0.90	$0.02	$0.01	$0.01	$0.01
Mistral: Mistral Small 3.1 24B	mistralai/mistral-small-3.1-24b-instruct	Text + image	128K	—	$0.53	$0.83	$0.04	$0.03	$0.03	$0.03
Mistral: Mistral Small 3.2 24B	mistralai/mistral-small-3.2-24b-instruct	Text + image	256K	—	$0.14	$0.38	$0.01	$0.01	$0.01	$0.01
Mistral: Mixtral 8x22B Instruct	mistralai/mixtral-8x22b-instruct	Text + file	65K	4.0	$3.00	$9.00	$0.20	$0.19	$0.17	$0.16
Mistral: Voxtral Small 24B 2507	mistralai/voxtral-small-24b-2507	Text + audio/file	32K	—	$0.15	$0.45	$0.00	$0.00	$0.00	$0.00
MoonshotAI: Kimi K2 0711	moonshotai/kimi-k2	Text	131K	19.7	$0.86	$3.45	$0.06	$0.06	$0.05	$0.05
MoonshotAI: Kimi K2 0905	moonshotai/kimi-k2-0905	Text	262K	24.0	$0.90	$3.75	$0.06	$0.06	$0.05	$0.05
MoonshotAI: Kimi K2 Thinking	moonshotai/kimi-k2-thinking	Text	262K	33.5	$0.90	$3.75	$0.06	$0.06	$0.05	$0.05
MoonshotAI: Kimi K2.5	moonshotai/kimi-k2.5	Text + image	262K	36.0	$0.67	$3.38	$0.05	$0.04	$0.04	$0.04
MoonshotAI: Kimi K2.6	moonshotai/kimi-k2.6	Text + image	262K	45.1	$0.84	$3.54	$0.06	$0.06	$0.05	$0.05
MoonshotAI: Kimi K2.7 Code	moonshotai/kimi-k2.7-code	Text + image	262K	43.0	$1.06	$5.25	$0.08	$0.07	$0.07	$0.06
MoonshotAI: Kimi K2.7 Code (batch)	moonshotai/kimi-k2.7-code:batch	Text + image	262K	43.0	$1.42	$6.00	$0.10	$0.09	$0.09	$0.08
MoonshotAI: Kimi K3	moonshotai/kimi-k3	Text + image/video	1.0M	59.7	$4.50	$22.50	$0.32	$0.30	$0.28	$0.26
Morph: Morph V3 Fast	morph/morph-v3-fast	Text	81K	—	$1.20	$1.80	$0.08	$0.08	$0.07	$0.07
Morph: Morph V3 Large	morph/morph-v3-large	Text	262K	—	$1.35	$2.85	$0.09	$0.09	$0.08	$0.07
Nex AGI: Nex-N2-Mini	nex-agi/nex-n2-mini	Text + image	262K	—	$0.04	$0.15	$0.00	$0.00	$0.00	$0.00
Nex AGI: Nex-N2-Pro	nex-agi/nex-n2-pro	Text + image	262K	41.7	$0.38	$1.50	$0.03	$0.02	$0.02	$0.02
Nous: Hermes 3 405B Instruct	nousresearch/hermes-3-llama-3.1-405b	Text	131K	—	$1.50	$1.50	$0.10	$0.10	$0.09	$0.08
Nous: Hermes 3 70B Instruct	nousresearch/hermes-3-llama-3.1-70b	Text	131K	4.8	$1.05	$1.05	$0.07	$0.07	$0.06	$0.06
Nous: Hermes 4 405B	nousresearch/hermes-4-405b	Text	131K	—	$1.50	$4.50	$0.10	$0.10	$0.09	$0.08
Nous: Hermes 4 70B	nousresearch/hermes-4-70b	Text	131K	—	$0.20	$0.60	$0.01	$0.01	$0.01	$0.01
NVIDIA: Nemotron 3 Nano 30B A3B	nvidia/nemotron-3-nano-30b-a3b	Text	262K	—	$0.07	$0.30	$0.01	$0.00	$0.00	$0.00
NVIDIA: Nemotron 3 Nano 30B A3B (free)	nvidia/nemotron-3-nano-30b-a3b:free	Text	256K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron 3 Nano Omni (free)	nvidia/nemotron-3-nano-omni-30b-a3b-reasoning:free	Text + audio/image/video	256K	15.0	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron 3 Super	nvidia/nemotron-3-super-120b-a12b	Text	1M	25.7	$0.13	$0.60	$0.01	$0.01	$0.01	$0.01
NVIDIA: Nemotron 3 Super (free)	nvidia/nemotron-3-super-120b-a12b:free	Text	262K	25.7	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron 3 Ultra	nvidia/nemotron-3-ultra-550b-a55b	Text	512K	38.3	$0.90	$5.40	$0.06	$0.06	$0.06	$0.05
NVIDIA: Nemotron 3 Ultra (batch)	nvidia/nemotron-3-ultra-550b-a55b:batch	Text	512K	38.3	$0.90	$5.40	$0.06	$0.06	$0.06	$0.05
NVIDIA: Nemotron 3 Ultra (free)	nvidia/nemotron-3-ultra-550b-a55b:free	Text	1M	38.3	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron 3.5 Content Safety (free)	nvidia/nemotron-3.5-content-safety:free	Text + image	128K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron 3.5 Lightning	nvidia/nemotron-3.5-lightning	Text	1M	23.6	$0.12	$0.30	$0.01	$0.01	$0.01	$0.01
NVIDIA: Nemotron 3.5 Lightning (free)	nvidia/nemotron-3.5-lightning:free	Text	1M	23.6	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron Nano 12B 2 VL (free)	nvidia/nemotron-nano-12b-v2-vl:free	Text + image/video	128K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
NVIDIA: Nemotron Nano 9B V2 (free)	nvidia/nemotron-nano-9b-v2:free	Text	128K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
OpenAI: GPT-3.5 Turbo	openai/gpt-3.5-turbo	Text	16K	3.2	$0.75	$2.25	$0.01	$0.01	$0.01	$0.01

OpenAI: GPT-3.5 Turbo (older v0613)	openai/gpt-3.5-turbo-0613	Text	4K	3.2	$1.50	$3.00	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-3.5 Turbo 16k	openai/gpt-3.5-turbo-16k	Text	16K	—	$4.50	$6.00	$0.07	$0.07	$0.06	$0.06
OpenAI: GPT-3.5 Turbo Instruct	openai/gpt-3.5-turbo-instruct	Text	4K	3.2	$2.25	$3.00	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-3.5 Turbo (batch)	openai/gpt-3.5-turbo:batch	Text	16K	3.2	$0.38	$1.12	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-4	openai/gpt-4	Text	8K	6.8	$45.00	$90.00	$0.40	$0.37	$0.34	$0.32
OpenAI: GPT-4 Turbo	openai/gpt-4-turbo	Text + image	128K	7.7	$15.00	$45.00	$1.05	$0.98	$0.91	$0.84
OpenAI: GPT-4 Turbo Preview	openai/gpt-4-turbo-preview	Text	128K	7.7	$15.00	$45.00	$1.05	$0.98	$0.91	$0.84
OpenAI: GPT-4 Turbo (batch)	openai/gpt-4-turbo:batch	Text + image	128K	7.7	$7.50	$22.50	$0.52	$0.49	$0.45	$0.42
OpenAI: GPT-4.1	openai/gpt-4.1	Text + file/image	1.0M	19.6	$3.00	$12.00	$0.21	$0.20	$0.18	$0.17
OpenAI: GPT-4.1 Mini	openai/gpt-4.1-mini	Text + file/image	1.0M	14.8	$0.60	$2.40	$0.04	$0.04	$0.04	$0.03
OpenAI: GPT-4.1 Mini (batch)	openai/gpt-4.1-mini:batch	Text + file/image	1.0M	14.8	$0.30	$1.20	$0.02	$0.02	$0.02	$0.02
OpenAI: GPT-4.1 Nano	openai/gpt-4.1-nano	Text + file/image	1.0M	9.6	$0.15	$0.60	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-4.1 Nano (batch)	openai/gpt-4.1-nano:batch	Text + file/image	1.0M	9.6	$0.07	$0.30	$0.01	$0.00	$0.00	$0.00
OpenAI: GPT-4.1 (batch)	openai/gpt-4.1:batch	Text + file/image	1.0M	19.6	$1.50	$6.00	$0.11	$0.10	$0.09	$0.08
OpenAI: GPT-4o	openai/gpt-4o	Text + file/image	128K	11.1	$3.75	$15.00	$0.26	$0.25	$0.23	$0.21
OpenAI: GPT-4o (2024-05-13)	openai/gpt-4o-2024-05-13	Text + file/image	128K	8.4	$7.50	$22.50	$0.52	$0.49	$0.45	$0.42
OpenAI: GPT-4o (2024-08-06)	openai/gpt-4o-2024-08-06	Text + file/image	128K	9.4	$3.75	$15.00	$0.26	$0.25	$0.23	$0.21
OpenAI: GPT-4o (2024-11-20)	openai/gpt-4o-2024-11-20	Text + file/image	128K	—	$3.75	$15.00	$0.26	$0.25	$0.23	$0.21
OpenAI: GPT-4o-mini	openai/gpt-4o-mini	Text + file/image	128K	6.7	$0.22	$0.90	$0.02	$0.01	$0.01	$0.01
OpenAI: GPT-4o-mini (2024-07-18)	openai/gpt-4o-mini-2024-07-18	Text + file/image	128K	—	$0.22	$0.90	$0.02	$0.01	$0.01	$0.01
OpenAI: GPT-4o-mini (batch)	openai/gpt-4o-mini:batch	Text + file/image	128K	6.7	$0.11	$0.45	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-4o (batch)	openai/gpt-4o:batch	Text + file/image	128K	11.1	$1.88	$7.50	$0.13	$0.12	$0.11	$0.11
OpenAI: GPT-5	openai/gpt-5	Text + file/image	400K	35.3	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
OpenAI: GPT-5 Codex (batch)	openai/gpt-5-codex:batch	Text + image	400K	37.0	$0.94	$7.50	$0.07	$0.06	$0.06	$0.05
OpenAI: GPT-5 Image	openai/gpt-5-image	Multimodal out	400K	—	$15.00	$15.00	$1.03	$0.96	$0.89	$0.82
OpenAI: GPT-5 Image Mini	openai/gpt-5-image-mini	Multimodal out	400K	—	$3.75	$3.00	$0.26	$0.24	$0.22	$0.21
OpenAI: GPT-5 Mini	openai/gpt-5-mini	Text + file/image	400K	25.8	$0.38	$3.00	$0.03	$0.03	$0.02	$0.02
OpenAI: GPT-5 Mini (batch)	openai/gpt-5-mini:batch	Text + file/image	400K	25.8	$0.19	$1.50	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-5 Nano	openai/gpt-5-nano	Text + file/image	400K	20.1	$0.07	$0.60	$0.01	$0.01	$0.00	$0.00
OpenAI: GPT-5 Nano (batch)	openai/gpt-5-nano:batch	Text + file/image	400K	20.1	$0.04	$0.30	$0.00	$0.00	$0.00	$0.00
OpenAI: GPT-5 Pro	openai/gpt-5-pro	Text + file/image	400K	—	$22.50	$180.00	$1.64	$1.53	$1.42	$1.31
OpenAI: GPT-5 Pro (batch)	openai/gpt-5-pro:batch	Text + file/image	400K	—	$11.25	$90.00	$0.82	$0.76	$0.71	$0.66
OpenAI: GPT-5.1	openai/gpt-5.1	Text + file/image	400K	37.5	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
OpenAI: GPT-5.1-Codex	openai/gpt-5.1-codex	Text + image	400K	35.6	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
OpenAI: GPT-5.1-Codex-Max	openai/gpt-5.1-codex-max	Text + image	400K	—	$1.88	$15.00	$0.14	$0.13	$0.12	$0.11
OpenAI: GPT-5.1-Codex-Mini	openai/gpt-5.1-codex-mini	Text + image	400K	31.3	$0.38	$3.00	$0.03	$0.03	$0.02	$0.02
OpenAI: GPT-5.1 (batch)	openai/gpt-5.1:batch	Text + file/image	400K	37.5	$0.94	$7.50	$0.07	$0.06	$0.06	$0.05
OpenAI: GPT-5.2	openai/gpt-5.2	Text + file/image	400K	43.3	$2.62	$21.00	$0.19	$0.18	$0.17	$0.15
OpenAI: GPT-5.2 Chat	openai/gpt-5.2-chat	Text + file/image	128K	43.3	$2.62	$21.00	$0.19	$0.18	$0.17	$0.15

OpenAI: GPT-5.2-Codex	openai/gpt-5.2-codex	Text + image	400K	41.2	$2.62	$21.00	$0.19	$0.18	$0.17	$0.15
OpenAI: GPT-5.2 Pro	openai/gpt-5.2-pro	Text + file/image	400K	—	$31.50	$252.00	$2.29	$2.14	$1.99	$1.83
OpenAI: GPT-5.2 Pro (batch)	openai/gpt-5.2-pro:batch	Text + file/image	400K	—	$15.75	$126.00	$1.15	$1.07	$0.99	$0.92
OpenAI: GPT-5.2 (batch)	openai/gpt-5.2:batch	Text + file/image	400K	43.3	$1.31	$10.50	$0.10	$0.09	$0.08	$0.08
OpenAI: GPT-5.3-Codex	openai/gpt-5.3-codex	Text + file/image	400K	45.5	$2.62	$21.00	$0.19	$0.18	$0.17	$0.15
OpenAI: GPT-5.4	openai/gpt-5.4	Text + file/image	1.1M	53.1	$3.75	$22.50	$0.27	$0.25	$0.23	$0.21
OpenAI: GPT-5.4 Image 2	openai/gpt-5.4-image-2	Multimodal out	272K	—	$12.00	$22.50	$0.83	$0.77	$0.72	$0.66
OpenAI: GPT-5.4 Mini	openai/gpt-5.4-mini	Text + file/image	400K	40.9	$1.12	$6.75	$0.08	$0.08	$0.07	$0.06
OpenAI: GPT-5.4 Mini (batch)	openai/gpt-5.4-mini:batch	Text + file/image	400K	40.9	$0.56	$3.38	$0.04	$0.04	$0.03	$0.03
OpenAI: GPT-5.4 Nano	openai/gpt-5.4-nano	Text + file/image	400K	39.7	$0.30	$1.88	$0.02	$0.02	$0.02	$0.02
OpenAI: GPT-5.4 Nano (batch)	openai/gpt-5.4-nano:batch	Text + file/image	400K	39.7	$0.15	$0.94	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-5.4 Pro	openai/gpt-5.4-pro	Text + file/image	1.1M	—	$45.00	$270.00	$3.22	$3.01	$2.79	$2.58
OpenAI: GPT-5.4 Pro (batch)	openai/gpt-5.4-pro:batch	Text + file/image	1.1M	—	$22.50	$135.00	$1.61	$1.50	$1.40	$1.29
OpenAI: GPT-5.4 (batch)	openai/gpt-5.4:batch	Text + file/image	1.1M	53.1	$1.88	$11.25	$0.13	$0.13	$0.12	$0.11
OpenAI: GPT-5.5	openai/gpt-5.5	Text + file/image	1.1M	56.3	$7.50	$45.00	$0.54	$0.50	$0.47	$0.43
OpenAI: GPT-5.5 Pro	openai/gpt-5.5-pro	Text + file/image	1.1M	—	$45.00	$270.00	$3.22	$3.01	$2.79	$2.58
OpenAI: GPT-5.5 Pro (batch)	openai/gpt-5.5-pro:batch	Text + file/image	1.1M	—	$22.50	$135.00	$1.61	$1.50	$1.40	$1.29
OpenAI: GPT-5.5 (batch)	openai/gpt-5.5:batch	Text + file/image	1.1M	56.3	$3.75	$22.50	$0.27	$0.25	$0.23	$0.21
OpenAI: GPT-5.6 Luna	openai/gpt-5.6-luna	Text + file/image	1.1M	52.3	$0.30	$1.80	$0.02	$0.02	$0.02	$0.02
OpenAI: GPT-5.6 Luna Pro	openai/gpt-5.6-luna-pro	Text + file/image	1.1M	—	$0.30	$1.80	$0.02	$0.02	$0.02	$0.02
OpenAI: GPT-5.6 Luna Pro (batch)	openai/gpt-5.6-luna-pro:batch	Text + file/image	1.1M	—	$0.15	$0.90	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-5.6 Luna (batch)	openai/gpt-5.6-luna:batch	Text + file/image	1.1M	52.3	$0.15	$0.90	$0.01	$0.01	$0.01	$0.01
OpenAI: GPT-5.6 Sol	openai/gpt-5.6-sol	Text + file/image	1.1M	60.9	$3.75	$22.50	$0.27	$0.25	$0.23	$0.21
OpenAI: GPT-5.6 Sol Pro	openai/gpt-5.6-sol-pro	Text + file/image	1.1M	—	$3.75	$22.50	$0.27	$0.25	$0.23	$0.21
OpenAI: GPT-5.6 Sol Pro (batch)	openai/gpt-5.6-sol-pro:batch	Text + file/image	1.1M	—	$1.88	$11.25	$0.13	$0.13	$0.12	$0.11
OpenAI: GPT-5.6 Sol (batch)	openai/gpt-5.6-sol:batch	Text + file/image	1.1M	60.9	$1.88	$11.25	$0.13	$0.13	$0.12	$0.11
OpenAI: GPT-5.6 Terra	openai/gpt-5.6-terra	Text + file/image	1.1M	56.6	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
OpenAI: GPT-5.6 Terra Pro	openai/gpt-5.6-terra-pro	Text + file/image	1.1M	—	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
OpenAI: GPT-5.6 Terra Pro (batch)	openai/gpt-5.6-terra-pro:batch	Text + file/image	1.1M	—	$1.50	$9.00	$0.11	$0.10	$0.09	$0.09
OpenAI: GPT-5.6 Terra (batch)	openai/gpt-5.6-terra:batch	Text + file/image	1.1M	56.6	$1.50	$9.00	$0.11	$0.10	$0.09	$0.09
OpenAI: GPT-5 (batch)	openai/gpt-5:batch	Text + file/image	400K	35.3	$0.94	$7.50	$0.07	$0.06	$0.06	$0.05
OpenAI: GPT Audio	openai/gpt-audio	Multimodal out	128K	—	$3.75	$15.00	$0.26	$0.25	$0.23	$0.21
OpenAI: GPT Audio Mini	openai/gpt-audio-mini	Multimodal out	128K	—	$0.90	$3.60	$0.06	$0.06	$0.05	$0.05
OpenAI: GPT Chat Latest	openai/gpt-chat-latest	Text + file/image	400K	60.9	$7.50	$45.00	$0.54	$0.50	$0.47	$0.43
OpenAI: gpt-oss-120b	openai/gpt-oss-120b	Text	131K	24.1	$0.04	$0.26	$0.00	$0.00	$0.00	$0.00
OpenAI: gpt-oss-20b	openai/gpt-oss-20b	Text	131K	15.2	$0.04	$0.20	$0.00	$0.00	$0.00	$0.00
OpenAI: gpt-oss-20b (free)	openai/gpt-oss-20b:free	Text	131K	15.2	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
OpenAI: gpt-oss-safeguard-20b	openai/gpt-oss-safeguard-20b	Text	131K	—	$0.11	$0.45	$0.01	$0.01	$0.01	$0.01
OpenAI: o1	openai/o1	Text + file/image	200K	23.9	$22.50	$90.00	$1.58	$1.48	$1.37	$1.27
OpenAI: o1-pro	openai/o1-pro	Text + file/image	200K	19.1	$225.00	$900.00	$15.84	$14.78	$13.73	$12.67

OpenAI: o1-pro (batch)	openai/o1-pro:batch	Text + file/image	200K	19.1	$112.50	$450.00	$7.92	$7.39	$6.86	$6.34
OpenAI: o1 (batch)	openai/o1:batch	Text + file/image	200K	23.9	$11.25	$45.00	$0.79	$0.74	$0.69	$0.63
OpenAI: o3	openai/o3	Text + file/image	200K	31.1	$3.00	$12.00	$0.21	$0.20	$0.18	$0.17
OpenAI: o3 Mini	openai/o3-mini	Text + file	200K	19.2	$1.65	$6.60	$0.12	$0.11	$0.10	$0.09
OpenAI: o3 Mini High	openai/o3-mini-high	Text + file	200K	15.7	$1.65	$6.60	$0.12	$0.11	$0.10	$0.09
OpenAI: o3 Mini High (batch)	openai/o3-mini-high:batch	Text + file	200K	15.7	$0.83	$3.30	$0.06	$0.05	$0.05	$0.05
OpenAI: o3 Mini (batch)	openai/o3-mini:batch	Text + file	200K	19.2	$0.83	$3.30	$0.06	$0.05	$0.05	$0.05
OpenAI: o3 Pro	openai/o3-pro	Text + file/image	200K	33.3	$30.00	$120.00	$2.11	$1.97	$1.83	$1.69
OpenAI: o3 Pro (batch)	openai/o3-pro:batch	Text + file/image	200K	33.3	$15.00	$60.00	$1.06	$0.99	$0.92	$0.84
OpenAI: o3 (batch)	openai/o3:batch	Text + file/image	200K	31.1	$1.50	$6.00	$0.11	$0.10	$0.09	$0.08
OpenAI: o4 Mini	openai/o4-mini	Text + file/image	200K	26.1	$1.65	$6.60	$0.12	$0.11	$0.10	$0.09
OpenAI: o4 Mini High	openai/o4-mini-high	Text + file/image	200K	26.1	$1.65	$6.60	$0.12	$0.11	$0.10	$0.09
OpenAI: o4 Mini High (batch)	openai/o4-mini-high:batch	Text + file/image	200K	26.1	$0.83	$3.30	$0.06	$0.05	$0.05	$0.05
OpenAI: o4 Mini (batch)	openai/o4-mini:batch	Text + file/image	200K	26.1	$0.83	$3.30	$0.06	$0.05	$0.05	$0.05
Free Models Router	openrouter/free	Text + image	200K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Perceptron: Perceptron Mk1	perceptron/perceptron-mk1	Text + image/video	32K	—	$0.22	$2.25	$0.01	$0.01	$0.01	$0.01
Perplexity: Sonar	perplexity/sonar	Text + image	127K	9.4	$1.50	$1.50	$0.10	$0.10	$0.09	$0.08
Perplexity: Sonar Deep Research	perplexity/sonar-deep-research	Text	128K	—	$3.00	$12.00	$0.21	$0.20	$0.18	$0.17
Perplexity: Sonar Pro	perplexity/sonar-pro	Text + image	200K	9.1	$4.50	$22.50	$0.32	$0.30	$0.28	$0.26
Perplexity: Sonar Pro Search	perplexity/sonar-pro-search	Text + image	200K	—	$4.50	$22.50	$0.32	$0.30	$0.28	$0.26
Perplexity: Sonar Reasoning Pro	perplexity/sonar-reasoning-pro	Text + image	128K	18.0	$3.00	$12.00	$0.21	$0.20	$0.18	$0.17
Poolside: Laguna S 2.1	poolside/laguna-s-2.1	Text	1.0M	—	$0.14	$0.27	$0.01	$0.01	$0.01	$0.01
Poolside: Laguna S 2.1 (free)	poolside/laguna-s-2.1:free	Text	262K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Poolside: Laguna XS 2.1	poolside/laguna-xs-2.1	Text	262K	—	$0.09	$0.18	$0.01	$0.01	$0.01	$0.00
Poolside: Laguna XS 2.1 (free)	poolside/laguna-xs-2.1:free	Text	262K	—	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Qwen2.5 72B Instruct	qwen/qwen-2.5-72b-instruct	Text	32K	9.4	$0.54	$0.60	$0.02	$0.02	$0.02	$0.01
Qwen: Qwen2.5 7B Instruct	qwen/qwen-2.5-7b-instruct	Text	32K	—	$0.15	$0.30	$0.01	$0.00	$0.00	$0.00
Qwen2.5 Coder 32B Instruct	qwen/qwen-2.5-coder-32b-instruct	Text	32K	6.9	$0.99	$1.50	$0.03	$0.03	$0.03	$0.03
Qwen: Qwen-Plus	qwen/qwen-plus	Text	1M	—	$0.39	$1.17	$0.03	$0.03	$0.02	$0.02
Qwen: Qwen Plus 0728	qwen/qwen-plus-2025-07-28	Text	1M	—	$0.39	$1.17	$0.03	$0.03	$0.02	$0.02
Qwen: Qwen Plus 0728 (thinking)	qwen/qwen-plus-2025-07-28:thinking	Text	1M	—	$0.39	$1.17	$0.03	$0.03	$0.02	$0.02
Qwen: Qwen2.5 VL 72B Instruct	qwen/qwen2.5-vl-72b-instruct	Text + image	128K	—	$1.20	$1.50	$0.08	$0.08	$0.07	$0.07
Qwen: Qwen3 14B	qwen/qwen3-14b	Text	131K	10.4	$0.18	$0.36	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 235B A22B	qwen/qwen3-235b-a22b	Text	131K	19.9	$0.68	$2.73	$0.05	$0.04	$0.04	$0.04
Qwen: Qwen3 235B A22B Instruct 2507	qwen/qwen3-235b-a22b-2507	Text	262K	18.4	$0.14	$0.83	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 235B A22B Thinking 2507	qwen/qwen3-235b-a22b-thinking-2507	Text	262K	19.9	$0.34	$3.45	$0.03	$0.02	$0.02	$0.02
Qwen: Qwen3 30B A3B	qwen/qwen3-30b-a3b	Text	131K	14.6	$0.20	$0.78	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 30B A3B Instruct 2507	qwen/qwen3-30b-a3b-instruct-2507	Text	262K	8.9	$0.07	$0.29	$0.01	$0.00	$0.00	$0.00
Qwen: Qwen3 30B A3B Thinking 2507	qwen/qwen3-30b-a3b-thinking-2507	Text	81K	14.6	$0.30	$3.60	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3 32B	qwen/qwen3-32b	Text	131K	11.4	$0.12	$0.42	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 8B	qwen/qwen3-8b	Text	131K	8.3	$0.18	$0.68	$0.01	$0.01	$0.01	$0.01

Qwen: Qwen3 Coder 480B A35B	qwen/qwen3-coder	Text	262K	18.2	$0.45	$1.50	$0.03	$0.03	$0.03	$0.03
Qwen: Qwen3 Coder 30B A3B Instruct	qwen/qwen3-coder-30b-a3b-instruct	Text	262K	13.6	$0.11	$0.42	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 Coder Flash	qwen/qwen3-coder-flash	Text	1M	—	$0.29	$1.46	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3 Coder Next	qwen/qwen3-coder-next	Text	262K	21.3	$0.18	$1.20	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 Coder Plus	qwen/qwen3-coder-plus	Text	1M	—	$0.98	$4.88	$0.07	$0.06	$0.06	$0.06
Qwen: Qwen3 Max	qwen/qwen3-max	Text	262K	24.5	$1.17	$5.85	$0.08	$0.08	$0.07	$0.07
Qwen: Qwen3 Max Thinking	qwen/qwen3-max-thinking	Text	262K	32.5	$1.17	$5.85	$0.08	$0.08	$0.07	$0.07
Qwen: Qwen3 Next 80B A3B Instruct	qwen/qwen3-next-80b-a3b-instruct	Text	262K	13.8	$0.14	$1.65	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 Next 80B A3B Thinking	qwen/qwen3-next-80b-a3b-thinking	Text	262K	16.9	$0.22	$1.80	$0.02	$0.02	$0.01	$0.01
Qwen: Qwen3 VL 235B A22B Instruct	qwen/qwen3-vl-235b-a22b-instruct	Text + image	262K	14.4	$0.32	$2.85	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3 VL 235B A22B Thinking	qwen/qwen3-vl-235b-a22b-thinking	Text + image	131K	20.9	$0.60	$6.00	$0.04	$0.04	$0.04	$0.04
Qwen: Qwen3 VL 30B A3B Instruct	qwen/qwen3-vl-30b-a3b-instruct	Text + image	262K	9.9	$0.20	$0.78	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 VL 30B A3B Thinking	qwen/qwen3-vl-30b-a3b-thinking	Text + image	262K	13.4	$0.30	$3.60	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3 VL 32B Instruct	qwen/qwen3-vl-32b-instruct	Text + image	131K	11.0	$0.16	$0.62	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 VL 8B Instruct	qwen/qwen3-vl-8b-instruct	Text + image	262K	8.2	$0.18	$0.68	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3 VL 8B Thinking	qwen/qwen3-vl-8b-thinking	Text + image	131K	10.5	$0.27	$3.15	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3.5-122B-A10B	qwen/qwen3.5-122b-a10b	Text + image/video	262K	32.8	$0.39	$3.12	$0.03	$0.03	$0.02	$0.02
Qwen: Qwen3.5-27B	qwen/qwen3.5-27b	Text + image/video	262K	34.6	$0.29	$2.34	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3.5-35B-A3B	qwen/qwen3.5-35b-a3b	Text + image/video	262K	29.9	$0.38	$1.88	$0.03	$0.02	$0.02	$0.02
Qwen: Qwen3.5 397B A17B	qwen/qwen3.5-397b-a17b	Text + image/video	262K	34.3	$0.58	$3.51	$0.04	$0.04	$0.04	$0.03
Qwen: Qwen3.5-9B	qwen/qwen3.5-9b	Text + image/video	262K	21.8	$0.15	$0.22	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3.5-Flash	qwen/qwen3.5-flash-02-23	Text + image/video	1M	—	$0.10	$0.39	$0.01	$0.01	$0.01	$0.01
Qwen: Qwen3.5 Plus 2026-02-15	qwen/qwen3.5-plus-02-15	Text + image/video	1M	—	$0.39	$2.34	$0.03	$0.03	$0.02	$0.02
Qwen: Qwen3.5 Plus 2026-04-20	qwen/qwen3.5-plus-20260420	Text + image/video	1M	—	$0.45	$2.70	$0.03	$0.03	$0.03	$0.03
Qwen: Qwen3.6 27B	qwen/qwen3.6-27b	Text + image/video	262K	37.7	$0.45	$3.00	$0.03	$0.03	$0.03	$0.03
Qwen: Qwen3.6 35B A3B	qwen/qwen3.6-35b-a3b	Text + image/video	262K	32.1	$0.21	$1.50	$0.02	$0.01	$0.01	$0.01
Qwen: Qwen3.6 Flash	qwen/qwen3.6-flash	Text + image/video	1M	—	$0.28	$1.69	$0.02	$0.02	$0.02	$0.02
Qwen: Qwen3.6 Max Preview	qwen/qwen3.6-max-preview	Text	262K	41.1	$1.54	$9.24	$0.11	$0.10	$0.10	$0.09
Qwen: Qwen3.6 Plus	qwen/qwen3.6-plus	Text + image/video	1M	40.5	$0.49	$2.92	$0.03	$0.03	$0.03	$0.03
Qwen: Qwen3.7 Flash	qwen/qwen3.7-flash	Text + image/video	1M	—	$0.04	$0.20	$0.00	$0.00	$0.00	$0.00
Qwen: Qwen3.7 Max	qwen/qwen3.7-max	Text	1M	46.7	$2.21	$6.64	$0.15	$0.14	$0.13	$0.12
Qwen: Qwen3.7 Plus	qwen/qwen3.7-plus	Text + image	1M	39.4	$0.48	$1.92	$0.03	$0.03	$0.03	$0.03
Qwen: Qwen3.8 2.4T A95B	qwen/qwen3.8-2.4t-a95b	Text	1.0M	57.7	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
Qwen: Qwen3.8 27B	qwen/qwen3.8-27b	Text + image/video	262K	52.0	$0.67	$4.80	$0.05	$0.05	$0.04	$0.04
Qwen: Qwen3.8 Max	qwen/qwen3.8-max	Text + image/video	1M	58.1	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
Reka Edge	rekaai/reka-edge	Text + image/video	16K	—	$0.15	$0.15	$0.00	$0.00	$0.00	$0.00
Reka Flash 3	rekaai/reka-flash-3	Text	65K	3.7	$0.15	$0.30	$0.01	$0.01	$0.01	$0.01
Relace: Relace Apply 3	relace/relace-apply-3	Text	256K	—	$1.27	$1.88	$0.09	$0.08	$0.08	$0.07
Relace: Relace Search	relace/relace-search	Text	256K	—	$1.50	$4.50	$0.10	$0.10	$0.09	$0.08
Sakana: Fugu Ultra	sakana/fugu-ultra	Text + image	1M	—	$7.50	$45.00	$0.54	$0.50	$0.47	$0.43

Sakana: Sakana Namazu	`sakana/sakana-namazu`	Text + file/image	262K	—	$1.42	$6.00	$0.10	$0.09	$0.09	$0.08
Sao10K: Llama 3 8B Lunaris	`sao10k/l3-lunaris-8b`	Text	8K	—	$0.06	$0.07	$0.00	$0.00	$0.00	$0.00
Sao10K: Llama 3.1 Euryale 70B v2.2	`sao10k/l3.1-euryale-70b`	Text	131K	—	$1.27	$1.27	$0.09	$0.08	$0.08	$0.07
Sao10K: Llama 3.3 Euryale 70B	`sao10k/l3.3-euryale-70b`	Text	131K	—	$0.98	$1.12	$0.07	$0.06	$0.06	$0.05
StepFun: Step 3.5 Flash	`stepfun/step-3.5-flash`	Text	262K	26.5	$0.15	$0.45	$0.01	$0.01	$0.01	$0.01
StepFun: Step 3.7 Flash	`stepfun/step-3.7-flash`	Text + image/video	262K	30.9	$0.30	$1.72	$0.02	$0.02	$0.02	$0.02
Tencent: Hunyuan A13B Instruct	`tencent/hunyuan-a13b-instruct`	Text	131K	—	$0.21	$0.86	$0.01	$0.01	$0.01	$0.01
Tencent: Hy3	`tencent/hy3`	Text	262K	42.2	$0.20	$0.79	$0.01	$0.01	$0.01	$0.01
Tencent: Hy3 preview	`tencent/hy3-preview`	Text	262K	34.4	$0.27	$0.90	$0.02	$0.02	$0.02	$0.02
TheDrummer: Cydonia 24B V4.1	`thedrummer/cydonia-24b-v4.1`	Text	131K	—	$0.45	$0.75	$0.03	$0.03	$0.03	$0.02
TheDrummer: Rocinante 12B	`thedrummer/rocinante-12b`	Text	65K	—	$0.38	$0.75	$0.02	$0.02	$0.02	$0.02
TheDrummer: Skyfall 36B V2	`thedrummer/skyfall-36b-v2`	Text	32K	—	$0.83	$1.20	$0.03	$0.03	$0.02	$0.02
TheDrummer: UnslopNemo 12B	`thedrummer/unslopnemo-12b`	Text	1.0M	—	$0.60	$0.60	$0.04	$0.04	$0.04	$0.03
Thinking Machines: Inkling	`thinkingmachines/inkling`	Text + audio/image	1.0M	42.3	$1.42	$6.07	$0.10	$0.09	$0.09	$0.08
Thinking Machines: Inkling Small	`thinkingmachines/inkling-small`	Text + audio/image	524K	41.2	$0.67	$1.80	$0.05	$0.04	$0.04	$0.04
Thinking Machines: Inkling (batch)	`thinkingmachines/inkling:batch`	Text + audio/image	524K	42.3	$1.50	$6.07	$0.11	$0.10	$0.09	$0.08
ReMM SLERP 13B	`undi95/remm-slerp-l2-13b`	Text	6K	—	$0.67	$0.98	$0.00	$0.00	$0.00	$0.00
Upstage: Solar Pro 3	`upstage/solar-pro-3`	Text	131K	14.5	$0.22	$0.90	$0.02	$0.01	$0.01	$0.01
Upstage: Solar Pro 4	`upstage/solar-pro4`	Text	524K	41.6	$0.04	$0.18	$0.00	$0.00	$0.00	$0.00
Writer: Palmyra X5	`writer/palmyra-x5`	Text	1.0M	—	$0.90	$9.00	$0.07	$0.06	$0.06	$0.05
SpaceXAI: Grok 4.20	`x-ai/grok-4.20`	Text + file/image	2M	38.0	$1.88	$3.75	$0.13	$0.12	$0.11	$0.10
SpaceXAI: Grok 4.20 Multi-Agent	`x-ai/grok-4.20-multi-agent`	Text + file/image	2M	—	$1.88	$3.75	$0.13	$0.12	$0.11	$0.10
SpaceXAI: Grok 4.3	`x-ai/grok-4.3`	Text + file/image	1M	37.9	$1.88	$3.75	$0.13	$0.12	$0.11	$0.10
SpaceXAI: Grok 4.5	`x-ai/grok-4.5`	Text + file/image	500K	55.8	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
SpaceXAI: Grok 4.6	`x-ai/grok-4.6`	Text + file/image	500K	60.9	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
SpaceXAI: Grok Build 0.1	`x-ai/grok-build-0.1`	Text + file/image	256K	40.7	$1.50	$3.00	$0.10	$0.10	$0.09	$0.08
Xiaomi: MiMo-V2.5	`xiaomi/mimo-v2.5`	Text + audio/image/video	1.1M	38.0	$0.21	$0.42	$0.01	$0.01	$0.01	$0.01
Xiaomi: MiMo-V2.5-Pro	`xiaomi/mimo-v2.5-pro`	Text	1.1M	42.9	$0.65	$1.30	$0.05	$0.04	$0.04	$0.04
Z.ai: GLM 4.5	`z-ai/glm-4.5`	Text	131K	19.7	$0.90	$3.30	$0.06	$0.06	$0.05	$0.05
Z.ai: GLM 4.5 Air	`z-ai/glm-4.5-air`	Text	131K	16.7	$0.20	$1.27	$0.01	$0.01	$0.01	$0.01
Z.ai: GLM 4.5V	`z-ai/glm-4.5v`	Text + image	65K	6.8	$0.90	$2.70	$0.06	$0.06	$0.05	$0.05
Z.ai: GLM 4.6	`z-ai/glm-4.6`	Text	204K	23.4	$0.75	$3.00	$0.05	$0.05	$0.05	$0.04
Z.ai: GLM 4.6V	`z-ai/glm-4.6v`	Text + image/video	131K	10.9	$0.45	$1.35	$0.03	$0.03	$0.03	$0.03
Z.ai: GLM 4.7	`z-ai/glm-4.7`	Text	204K	34.5	$0.60	$2.62	$0.04	$0.04	$0.04	$0.03
Z.ai: GLM 4.7 Flash	`z-ai/glm-4.7-flash`	Text	202K	23.3	$0.09	$0.60	$0.01	$0.01	$0.01	$0.01
Z.ai: GLM 5	`z-ai/glm-5`	Text	204K	40.6	$0.90	$2.88	$0.06	$0.06	$0.05	$0.05
Z.ai: GLM 5 Turbo	`z-ai/glm-5-turbo`	Text	202K	39.1	$1.80	$6.00	$0.13	$0.12	$0.11	$0.10
Z.ai: GLM 5.1	`z-ai/glm-5.1`	Text	204K	41.0	$1.45	$4.55	$0.10	$0.09	$0.09	$0.08
Z.ai: GLM 5.2	`z-ai/glm-5.2`	Text	1.0M	52.6	$1.45	$4.55	$0.10	$0.09	$0.09	$0.08
Z.ai: GLM 5.2 (batch)	`z-ai/glm-5.2:batch`	Text	512K	52.6	$2.10	$6.60	$0.15	$0.14	$0.13	$0.12
Z.ai: GLM 5.2 (free)	`z-ai/glm-5.2:free`	Text	256K	52.6	$0.00	$0.00	$0.00	$0.00	$0.00	$0.00
Z.ai: GLM 5.3	`z-ai/glm-5.3`	Text	1.0M	59.5	$2.10	$6.60	$0.15	$0.14	$0.13	$0.12
Z.ai: GLM 5V Turbo	`z-ai/glm-5v-turbo`	Text + image/video	202K	35.3	$1.80	$6.00	$0.13	$0.12	$0.11	$0.10
Anthropic: Claude Fable Latest	`~anthropic/claude-fable-latest`	Text + file/image	1M	62.1	$15.00	$75.00	$1.07	$0.99	$0.92	$0.85

Anthropic Claude Haiku Latest	`~anthropic/claude-haiku-latest`	Text + file/image	200K	29.9	$1.50	$7.50	$0.11	$0.10	$0.09	$0.09
Anthropic: Claude Opus Latest	`~anthropic/claude-opus-latest`	Text + file/image	1M	63.1	$7.50	$37.50	$0.53	$0.50	$0.46	$0.43
Anthropic Claude Sonnet Latest	`~anthropic/claude-sonnet-latest`	Text + file/image	1M	55.3	$3.00	$15.00	$0.21	$0.20	$0.18	$0.17
DeepSeek V4 Flash Latest	`~deepseek/deepseek-v4-flash-latest`	Text	1.3M	51.8	$0.11	$0.23	$0.01	$0.01	$0.01	$0.01
Google Gemini Flash Latest	`~google/gemini-flash-latest`	Text + audio/file/image/video	1.0M	56.0	$0.56	$2.81	$0.04	$0.04	$0.03	$0.03
Google Gemini Pro Latest	`~google/gemini-pro-latest`	Text + audio/file/image/video	1.0M	47.7	$3.00	$18.00	$0.21	$0.20	$0.19	$0.17
MoonshotAI Kimi Latest	`~moonshotai/kimi-latest`	Text + image/video	1.0M	59.7	$3.90	$19.50	$0.28	$0.26	$0.24	$0.22
OpenAI GPT Latest	`~openai/gpt-latest`	Text + file/image	1.1M	60.9	$3.75	$22.50	$0.27	$0.25	$0.23	$0.21
OpenAI GPT Mini Latest	`~openai/gpt-mini-latest`	Text + file/image	400K	40.9	$1.12	$6.75	$0.08	$0.08	$0.07	$0.06
xAI: Grok Latest	`~x-ai/grok-latest`	Text + file/image	500K	7.3	$3.00	$9.00	$0.21	$0.20	$0.18	$0.17
Z.ai: GLM Latest	`~z-ai/glm-latest`	Text	1.0M	59.5	$2.10	$6.60	$0.15	$0.14	$0.13	$0.12

Rates are pulled at build time and can change.

Aardvark Extras

Most built-in tags wrap a Mantine component one-to-one. The tags on this page are different: they are **aardvark-native** components with no direct Mantine equivalent — purpose-built widgets for documentation sites, each a single tag you drop straight into Markdown.

- API Fields — hand-authored API parameter and response-field rows (`{% field %}`) for references not driven by an OpenAPI spec.
- Banner — a full-width announcement row across the top of the page, in your site's primary color.
- Changelog — a timeline of changes read from a YAML data file, filterable by tag, with an RSS feed.
- CodeGroup — several fenced code blocks shown as language / file tabs, each with its own copy button; the chosen language syncs across every code group on the page.
- Component libraries — pull extra React libraries into your theme and address them with `{% component('library', 'Name') %}`.
- Examples — side-by-side request/response panels for API docs: titled, syntax-highlighted code blocks with per-block copy/download, and a tab strip for multiple languages or status codes.
- Gitfolder — embed a public GitHub repo folder: file tree, highlighted source, inline images, a Markdown/SVG preview toggle, and a Download .zip button.
- Include — splice a shared Markdown partial into a page, varied by the including page's front matter.
- Map — an embedded OpenFreeMap / MapLibre map with one pin per location, placed by address or coordinates.
- OpenAPI — splice a single endpoint from an OpenAPI spec inline next to your prose, or render a whole spec as a full reference.
- Panel — a supplementary side panel that floats beside the main content on a wide viewport and stacks below it on a narrow one.
- Prompt — a copyable AI-prompt block: the prompt text with a copy button plus "Open in ChatGPT / Claude / Cursor" deep-link buttons that pre-fill the prompt.
- Target — name a region of a page so Content Reach reports on it as its own section, alongside the ones your headings already create.
- Taxonomy — render tagged member pages as a changelog timeline, a knowledge base, or a blog-style article list, with tag filtering and an RSS feed for changelog and article listings.
- Update — an inline release-note / changelog entry on a timeline rail: a version or date label on the left, the Markdown body beside it.
- Visibility — show or hide a block of Markdown depending on whether a human is reading the HTML page or an AI agent is reading its Markdown twin.

API Fields

A **built-in** tag for documenting an API **by hand** — when there is no OpenAPI spec driving the page. `{% field %}` renders one labeled row: the field **name** (monospace), its **type**, `required` / `deprecated` badges, an optional **default**, an optional **location** badge (query / path / body / header / cookie), and a **Markdown description** written as the block body.

One tag covers both request parameters and response fields, because the two render identically. The only request-only affordance is the `location` badge — and a response field, being just a key in the returned payload, simply omits it.

This is for prose you write yourself. If you already have an OpenAPI spec, reach for **OpenAPI** instead — it renders parameter and response tables straight from the spec, with a "Try it now" form. `field` is the spec-free counterpart: drop one row at a time into a guide.

Use it as `{% field %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'field', …)`.

What `location` means

`location` is the one attribute that makes a row a **request parameter**. It answers the question a name and a type can't: where does this value go in the HTTP request?

`location`	Where the value rides	Example
`query`	The URL query string	`GET /pets?limit=20`
`path`	A URL path segment	`GET /pets/{petId}`
`header`	A request header	`X-Api-Version: 2`
`cookie`	The `Cookie` header	`Cookie: session=abc`
`body`	The request body	`{"limit": 20}`

It is the hand-authored mirror of OpenAPI's `in` field: a `field` with `location="query"` renders exactly what `{% openapi %}` shows for `in: query`. A **response** field has no request location, so you just leave `location` off.

Request parameters

`{% field %}` takes the field `name` (the only required attribute), an optional `type`, the `required` and `deprecated` flags, an optional `default`, and an optional `location`. The block body is the description — full Markdown. Close it with `{% endField %}`.

`query` string [required] [query]

The search term to match against. Case-insensitive; supports `AND` / `OR` operators.

`limit` integer [query] default: `20`

Maximum number of results to return. Must be between **1** and **100**.

`petId` string [required] [path]

The unique identifier of the pet, from a prior list response.

`X-Api-Version` string [deprecated] [header]

Pins the request to a legacy API version. **Deprecated** — omit it to use the current version.

Source: Markdown

```
{% field name="query" type="string" required=true location="query" %}
The search term to match against. Case-insensitive; supports `AND` / `OR` operators.
{% endField %}

{% field name="limit" type="integer" default="20" location="query" %}
Maximum number of results to return. Must be between **1** and **100**.
{% endField %}

{% field name="petId" type="string" required=true location="path" %}
The unique identifier of the pet, from a prior list response.
{% endField %}

{% field name="X-Api-Version" type="string" deprecated=true location="header" %}
Pins the request to a legacy API version. **Deprecated** — omit it to use the current
version.
{% endField %}
```

Source: Python

```
component('aardvark', 'field', name='query', type='string',
          required=True, location='query',
          children='The search term to match against. Case-insensitive.')
component('aardvark', 'field', name='limit', type='integer',
          default='20', location='query',
          children='Maximum number of results to return. Must be between **1** and
**100**.')
```

Response fields

A response field is the same tag with **no `location`** — there is nowhere in a request for it to go. Everything else works identically.

`id` string [required]

The unique identifier for the object, prefixed with its type (e.g. `pet_8xKq…`).

`object` string [required]

The type of object. Always **`pet`** for this endpoint.

`tags` array default: `[]`

The labels attached to this pet. Each entry has an `id` and a `name`.

`legacyStatus` integer [deprecated]

The old numeric status code. **Deprecated** in favor of the string `status` field.

Source: Markdown

```
{% field name="id" type="string" required=true %}
The unique identifier for the object, prefixed with its type (e.g. `pet_8xKq…`).
{% endField %}

{% field name="object" type="string" required=true %}
The type of object. Always **`pet`** for this endpoint.
{% endField %}

{% field name="tags" type="array" default="[]" %}
The labels attached to this pet. Each entry has an `id` and a `name`.
{% endField %}

{% field name="legacyStatus" type="integer" deprecated=true %}
The old numeric status code. **Deprecated** in favor of the string `status` field.
{% endField %}
```

Source: Python

```
component('aardvark', 'field', name='id', type='string', required=True,
          children='The unique identifier for the object.')
component('aardvark', 'field', name='tags', type='array', default='[]',
          children='The labels attached to this pet. Each entry has an `id` and a `name`.')
```

Request vs response fields

One tag serves both. A **request parameter** carries a `location` (`query`, `path`, `body`, `header`, or `cookie` — OpenAPI's `in`, where the value rides in the request); a **response field** is just a key in the returned payload, so it omits `location` and shows no location badge. Nothing else differs — the two render identically.

```
{% field name="petId" type="integer" required=true location="path" %}
A request parameter — the `location` badge marks where it rides.
{% endField %}

{% field name="id" type="string" %}
A response field — no location, no badge.
```

```
{% endField %}
```

Rich descriptions

The body is full Markdown, so a field description can hold emphasis, links, inline code, and lists — handy for enumerating allowed values or nested shapes:

`status` string [required]

The lifecycle state of the pet. One of:

- `available` — ready to be adopted.
- `pending` — an adoption is in progress.
- `sold` — no longer available.

See the status guide for the full state machine.

Source: Markdown

```
{% field name="status" type="string" required=true %}
The lifecycle state of the pet. One of:

- `available` — ready to be adopted.
- `pending` — an adoption is in progress.
- `sold` — no longer available.

See the status guide for the full state machine.
{% endField %}
```

Attributes

Attribute	Valid values	Description
`name`	Any string (**required**)	The field name, shown in monospace. The row's anchor.
`type`	Any string	The field's type (`string`, `integer`, `array`, ...), shown as dimmed mono text.
`required`	`true` / `false` (default)	Shows a `required` badge when true.
`deprecated`	`true` / `false` (default)	Shows a `deprecated` badge when true.
`default`	Any string	Shows `default: <value>` after the badges.
`location`	`query`, `path`, `body`, `header`, `cookie` (or any string)	A color-coded badge showing where the parameter is sent in the request. Omit it for a response field.
`attr`	`{…}`	Raw HTML attributes forwarded onto the rendered field row (see below).
(body)	Markdown	The description, written between the open and close tags (`children=` from Python).

CSS Selectors

Every row renders the same `ApiField` island, so one wrapper class targets them all.

```
[data-aardvark-island="ApiField"] {
  /* style every field row on the page */
}

[data-aardvark-island="ApiField"] .aardvark-api-field-name {
  /* the monospace field name */
}

[data-aardvark-island="ApiField"] .aardvark-api-field-body {
  /* the Markdown description */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including an `id` for deep-linking) straight onto the rendered field row.

`apiKey` string [required] [header]

Your secret API key. This row carries `id="field-api-key"` so you can link straight to it.

Source: Markdown

```
{% field name="apiKey" type="string" required=true location="header" attr={'id':
'field-api-key'} %}
Your secret API key. This row carries `id="field-api-key"` so you can link straight to it.
{% endField %}
```

Source: Python

```
component('aardvark', 'field', name='apiKey', type='string', required=True,
          location='header', children='Your secret API key.',
          attr={'id': 'field-api-key'})
```

Banner

A **banner** is a full-width announcement row pinned across the very top of the page, above the header, filled with your site's **primary color**. The colored row at the very top of this page is a live banner. Its text is ordinary **Markdown**, so **bold**, `code`, and links all work.

Unlike the other built-ins, the banner isn't a tag you write in the body — there's nothing to place inside your content. You turn it on from **configuration**: once site-wide, or per page in front matter.

Site-wide

Set a `banner:` string in `aardvark.config.yaml` and it shows on every page:

```
banner: "**Heads up:** scheduled maintenance this Saturday. [Status page](/status/)."
```

Per page

A page can set its own `banner:` in front matter. It **overrides** the site-wide banner on that page only:

```
---
title: "Release notes"
banner: "**Version 2.0** is out — see what changed below."
---
```

To **hide** the site-wide banner on a single page, set `banner: false`:

```
---
title: "Checkout"
banner: false
---
```

Behavior

- **Color** is always your theme's primary color, in both light and dark mode.
- **Markdown**, rendered inline — **bold**, italics, `code`, and links. (It's plain text, not a place for tags or other components.)
- **Dismissible** — a × button closes it; it stays closed for that visitor until you change the text. Editing the announcement brings it back for everyone.
- **Scrolls away** — the banner sits at the top and scrolls off as the reader moves down the page, while the header stays pinned.

CSS Selectors

The banner is rendered straight into the theme markup (it isn't a hydrated island), so target its own class names — the bar, its inner content row, and the dismiss button.

```
.aardvark-banner        /* the full-width bar */
.aardvark-banner-inner  /* the centered content row */
.aardvark-banner-close  /* the × dismiss button */
```

Injecting Attributes

The banner isn't a body tag — there's no `{% banner %}` invocation to take a raw `attr={…}` channel. It's turned on from configuration (`banner:` in `aardvark.config.yaml` or page front matter), and styled through the CSS classes above.

Changelog

A **built-in** tag that turns a YAML data file of changes into a Mantine Timeline — newest first, each entry with a date, an optional release **version** badge, and a **Markdown** description of any length. A cloud of tag badges lets readers **filter** the timeline; in a wide layout the cloud sits in a sticky right-hand column, otherwise above the entries. Each changelog also publishes an **RSS feed** (linked from the icon above it) and — in a normal-width layout — adds every change to the page's **On this page** list. This site's live Changelog tab is now **taxonomy-driven** — the same timeline, fed from member articles by the `{% taxonomy %}` tag — while `{% changelog %}` remains fully supported for YAML-file changelogs like the live examples on this page.

Usage

Point the tag at a YAML data file (path relative to your site root). Everything else is optional:

```
{% changelog "data/changelog.yaml" %}
```

renders, live (capped to the four most recent entries here):

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Standardized code-block Copy & Download buttons — May 22, 2026

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

Index OpenAPI descriptions for search — May 14, 2026

Operation and schema descriptions from your OpenAPI spec are now part of the search index, so a reader searching for an endpoint's behavior lands on the right reference section — not just pages that happen to mention it in prose.

Click a tag to filter; click more than one to combine them — an entry shows if it matches **any** active tag. Click an active tag again, or **Clear**, to reset.

The data file

The data file is a **YAML list** of changes. Each change needs a `title`, a `date`, and a `description` (Markdown of any length); `version` and `tags` are optional. A `date` may include a **time** — the full timestamp drives the sort, so two changes on the same day order correctly:

```
- title: Generate the CLI reference from `vark --help`
  date: 2026-05-28 16:30            # a time is optional; it refines the sort
  version: "0.9.0"                  # optional — a release badge beside the date
  tags: [cli, docs]                 # optional — populate the filter cloud
  description: |                    # required — rendered as Markdown (links welcome)
    Generated straight from `vark --help`. See the [CLI reference](/cli/).

- title: Build-time accessibility contrast audit
  date: 2026-04-19
  tags: [a11y, build]
  description: |
    A non-fatal **WCAG contrast audit** over your theme colors.
```

Field	Required	Notes
`title`	yes	The change's headline.
`date`	yes	`YYYY-MM-DD`, optionally with a time (`2026-05-28 16:30`). Entries render newest first by the full timestamp.
`description`	yes	Markdown of any length — lists, code, links, emphasis.
`version`	no	A release marker shown as a badge beside the date.
`tags`	no	A list of labels (or a comma-separated string); these populate the filter cloud.

Filtering and limiting

Only one tag. Pass `tags=` to render just the changes carrying one of those tags. The list is then already filtered, so no tag cloud is drawn:

```
{% changelog "data/changelog.yaml" tags="build" %}
```

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Build-time accessibility contrast audit — April 19, 2026

`vark build` now runs a non-fatal **WCAG color-contrast audit** over your theme colors (light and dark) and reports any pair that falls below the configured level, so regressions surface before they ship. Read more under Accessibility.

Multi-language sites — March 21, 2026

Serve translated content from per-language directories with an automatic language picker. `vark build --translate` fills missing or changed pages with a model, grounded in a reusable translation memory.

Only the last few. `limitEntries=N` caps how many entries render (after the newest-first sort); `limitDays=N` instead keeps only changes from the last N days, counted from the build date. Showing the three most recent:

```
{% changelog "data/changelog.yaml" limitEntries=3 %}
```

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Standardized code-block Copy & Download buttons — May 22, 2026

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

Because this page uses a normal (non-wide) layout, those three changes are also linked from the **On this page** list on the right — that's the default. Add `toc=false` to opt out (the other examples on

this page do, to keep this page's contents about the docs).

Combining same-day changes

Add `combineByDay` to merge every change from one calendar day into a single timeline entry — the date shows once, with each change listed beneath it. (Add times to your dates when you don't want this, so same-day changes stay separate but still sort correctly.)

```
{% changelog "data/changelog.yaml" combineByDay limitEntries=2 %}
```

Generate the CLI reference from `vark --help` — May 28, 2026

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Standardized code-block Copy & Download buttons — May 22, 2026

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

Feeds

Every changelog publishes an **RSS feed** at `feed.xml` beside the page, with one item per change. (The Changelog tab's `/changelog/feed.xml` is published the same way by its taxonomy listing.) The **RSS icon** above the timeline links to it, and the page advertises the feed in its `<head>` so feed readers discover it automatically. No configuration needed.

Options

Attribute	Effect
`"…"` (first argument)	Path to the YAML data file, relative to the site root (**required**).
`wide`	Lay the tag cloud out in a sticky right-hand column. **Requires a wide layout mode** — `mode: wide`, `full`, or `uncapped` in the page front matter (see Modes); otherwise the build fails with a clear error. Without `wide`, the cloud renders above the timeline.
`tags="a,b"`	Show only changes carrying one of these tags; suppresses the tag cloud.
`combineByDay`	Merge all of a day's changes into one timeline entry.
`toc=false`	Don't add the changes to the page's "On this page" list (added by default in a non-wide layout; wide layouts hide that list).
`limitDays=90`	Show only changes from the last N days, counted from the build date.
`limitEntries=20`	Show at most N entries (after the newest-first sort and any combining).

For the full **wide** layout — the timeline beside a sticky tag cloud — set a wide mode (`mode: wide` or `full`) and add `wide`, exactly as the Changelog tab does with its taxonomy-driven equivalent:

```
{% changelog "data/changelog.yaml" wide %}
```

CSS Selectors

The feed mounts inside an island wrapper carrying `data-aardvark-island="Changelog"` and renders its own class names — target the feed, each entry, its meta line, and a change row.

```
[data-aardvark-island="Changelog"]  /* the island wrapper */
.aardvark-changelog                 /* the whole widget */
.aardvark-changelog-feed            /* the entry timeline */
.aardvark-changelog-item            /* a single dated entry */
.aardvark-changelog-change          /* one change line */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered changelog root. (Style it through the CSS parts above, and shape the feed with the documented attributes: `tags`, `wide`, `combineByDay`, `limitDays`, `limitEntries`.)

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Standardized code-block Copy & Download buttons — May 22, 2026

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

Source: Markdown

```
{% changelog "data/changelog.yaml" limitEntries=3 toc=false attr={'data-analytics':
'release-feed', 'aria-label': 'Changelog'} %}
```

Source: Python

```
component('aardvark', 'changelog', 'data/changelog.yaml', limitEntries=3, toc=False,
          attr={'data-analytics': 'release-feed', 'aria-label': 'Changelog'})
```

CodeGroup

`{% codeGroup %}` wraps a run of fenced code blocks and shows them as **language / file tabs**, each with its own copy button. Reach for it when the same thing is shown several ways — a request in `curl`, Python, and JavaScript; a config in YAML and TOML; a file across a few languages — so the reader picks the tab they care about instead of scrolling past the rest.

Every fenced block inside becomes one tab. The tab label comes from the block's language (`json` → **JSON**), or from an explicit `title="…"` on the fence when you want a filename or a custom name. The code is highlighted at build time with the same highlighter the site's ``` fenced blocks use, so a tab looks exactly like an ordinary code block — just grouped.

Use it as `{% codeGroup %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'codeGroup', …)`. Close it with `{% endCodeGroup %}`.

CodeGroup or Tabs?

A code group is the same tab widget as **Tabs** — same variants, same sliding underline, same crossfade — specialised for code. Reach for **CodeGroup** when the tabs are the same thing in different languages or files: you get automatic language labels, a copy / download button per tab, build-time highlighting, and a language choice that **syncs across every group on the page and persists**. Reach for **Tabs** when the panels are arbitrary Markdown — prose, lists, nested components, mixed content — that you label yourself and that should stay independent.

By language

Give each way its own fenced block and the languages become the tabs:

bash

```bash
curl https://api.example.com/v1/users \
  -H "Authorization: Bearer $TOKEN"
```

python

```python
import requests

requests.get(
    "https://api.example.com/v1/users",
    headers={"Authorization": f"Bearer {token}"},
)
```

javascript

```javascript
await fetch("https://api.example.com/v1/users", {
  headers: { Authorization: `Bearer ${token}` },
});
```

Source: Markdown

````
{% codeGroup %}
```bash
curl https://api.example.com/v1/users \
````
````

````
  -H "Authorization: Bearer $TOKEN"
```
```python
import requests

requests.get(
    "https://api.example.com/v1/users",
    headers={"Authorization": f"Bearer {token}"},
)
```
```javascript
await fetch("https://api.example.com/v1/users", {
  headers: { Authorization: `Bearer ${token}` },
});
```
{% endCodeGroup %}
````

Named tabs with `title`

Add `title="…"` to a fence to label the tab yourself — a filename is the common case. The language still drives the highlighting; only the label changes:

package.json

```
{
  "name": "my-app",
  "version": "1.0.0"
}
```

pyproject.toml

```
[project]
name = "my-app"
version = "1.0.0"
```

Source: Markdown

````
{% codeGroup %}
```json title="package.json"
{
  "name": "my-app",
  "version": "1.0.0"
}
```
```toml title="pyproject.toml"
[project]
name = "my-app"
version = "1.0.0"
```
{% endCodeGroup %}
````
````

Choosing the opening language

Every code group on a page opens on the **same language** (see Syncing the language across groups below) — by default the first code group's first tab. To make a page open on a specific language, set `defaultValue` (a tab's label) on the **first** code group; it seeds the language every group then opens on:

````
{% codeGroup defaultValue="Python" %}
```bash
echo "shell is first in source…"
```
```python
print("…but the page opens on Python")
```
{% endCodeGroup %}
````

`defaultValue` only sets the first, pre-choice paint: once a reader picks a language it's remembered (below) and that wins. On a later group it has no visible effect — the page's shared language has already applied — so reach for it on the first group only.

Syncing the language across groups

Code groups on a page share **one language choice**, from the very first paint: they all open on the same language, and picking **Python** in any group switches every other group that has a Python tab to it at once. The choice sticks, too — the next page you open starts on Python. It's remembered per-reader in the browser, so a `curl`-first reader and a Python-first reader each read the docs in the language they picked, across the whole site.

Try it — click a language in either group and watch the other follow:

bash

```bash
curl https://api.example.com/v1/pets
```

python

```python
requests.get("https://api.example.com/v1/pets")
```

javascript

```javascript
await fetch("https://api.example.com/v1/pets");
```

bash

```bash
curl https://api.example.com/v1/pets/1 -X DELETE
```

python

```python
requests.delete("https://api.example.com/v1/pets/1")
```

javascript

```javascript
await fetch("https://api.example.com/v1/pets/1", { method: "DELETE" });
```

The match is by the **tab label** — the language name, or a `title="…"` if you set one. A group that doesn't have the picked language just stays on whatever it's showing.

Variants

A code group is the same tab widget as Tabs, so it takes the same `variant`. It defaults to **`pills`** (the filled buttons above); pass `variant="default"` (or `"outline"`) for the **sliding underline** instead — the same animated indicator `{% tabs %}` uses:

bash

```
npm install my-app
```

pnpm

```
pnpm add my-app
```

yarn

```
yarn add my-app
```

Attributes

Attribute	Valid values	Description
(body)	Fenced code blocks	Each ``` (or ~~~) fenced block becomes one tab, in source order. Written between `{% codeGroup %}` and `{% endCodeGroup %}`.
`defaultValue`	A tab label	On the **first** code group of a page, the language every group opens on; on a later group it only shows before the shared language applies. Overridden once a reader picks a language (remembered site-wide). Defaults to the first tab. Match the label — the language name (e.g. `JavaScript`) or a fence's `title`.
`variant`	`pills`, `default`, `outline`	The tab style, shared with Tabs. Defaults to `pills`; `default` / `outline` give the sliding underline.
`attr`	`{…}`	Raw HTML attributes forwarded onto the widget's root element (see below).

Per-tab, on the fence itself:

Fence syntax	Description
```` ```lang ````	The language — drives both syntax highlighting and, unless overridden, the tab label (`json` → **JSON**). A fence with no language becomes a plain **Text** tab.

```` ```lang title="name" ````	An explicit tab label, typically a filename. The language still drives highlighting.

Use a `~~~` fence (tildes) for a block whose own code contains a ``` ``` run — the tab body is taken verbatim.

CSS Selectors

Each code group carries `data-aardvark-island="CodeGroup"` on its wrapper, and the rendered Mantine `Tabs` exposes its parts as `mantine-Tabs-*` classes. The per-tab code reuses the site's shared `.aardvark-code-block` / `.aardvark-code-actions` markup, so it styles identically to an on-page fenced block.

```
[data-aardvark-island="CodeGroup"] {
  /* style every code group on the page */
}

.aardvark-code-group .mantine-Tabs-tab {
  /* a language / file tab */
}

.aardvark-code-group .aardvark-code-block {
  /* the active tab's code block */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered root element.

bash

```
npm install my-app
```

yarn

```
yarn add my-app
```

Source: Markdown

````
{% codeGroup attr={'id': 'install-group'} %}
```bash
npm install my-app
```
```bash title="yarn"
yarn add my-app
```
{% endCodeGroup %}
````

Component libraries

Every Mantine component is available out of the box, addressed with the library name first — `{% component('mantine', 'Button') %}`. And a **theme isn't limited to Mantine:** it can pull in any React component library — Stripe Elements, a charting kit, an icon set — and expose it through the same `{% component() %}` tag.

The library name is what keeps two libraries that both export, say, `Tooltip` from colliding — a charting kit's `Tooltip` and Mantine's stay distinct because each is reached through its own key:

```
{% component('mantine', 'Button', color='blue') %}      {# Mantine — the built-in library #}
{% component('stripe', 'PaymentElement') %}             {# a library this theme declared #}
```

For the built-in sources — Mantine, the native components, and your project snippets — you can drop the prefix: `{% component('Button') %}` is shorthand for `{% component('mantine', 'Button') %}`. That shorthand is **always** unambiguous — a theme library is only ever reachable through its name, so a library's `Button` can never capture the bare one.

Declaring a library

A theme declares its extra libraries in a **`theme.yaml`** beside its templates (`themes/<name>/theme.yaml`). This site pulls in Stripe's React SDK:

```
# themes/vark/theme.yaml
componentLibraries:
  stripe:
    package: "@stripe/react-stripe-js"   # the npm import specifier (must be `npm install`ed)
    components: [CardElement, PaymentElement, LinkAuthenticationElement]   # omit to auto-discover
```

The **key** (`stripe`) is the first argument you pass to `{% component() %}`; the **`package`** is what gets imported and bundled. Because the two are separate, the key is just a friendly handle. Keys are lowercase-kebab and may not be `mantine`, `aardvark`, or `snippet` (those name the built-in sources).

The one piece a library can't hand you

Stripe's Element components — `PaymentElement`, `CardElement` — render real, PCI-compliant card fields inside a cross-origin iframe. But they only work inside Stripe's `<Elements>` provider, which needs a live Stripe instance from `loadStripe('pk_…')` — a call that returns a Promise and injects Stripe.js. That's runtime JavaScript, not a value you can pass as a Markdown prop. So the **provider** is one small snippet; everything inside it is addressed from the library.

The provider is also where you set Stripe's **Appearance** — here it follows the page's light/dark toggle so the Payment Element keeps its contrast (`useColorScheme()` reads `<html data-mantine-color-scheme>`). This is the real file in full — it's SSR-safe (`loadStripe` only runs in the browser; `options` is memoized) and `deferred` is what the storefront below flips to swap the

Payment Element for a card field:

```
// snippets/StripeProvider.jsx
import { useEffect, useMemo, useState } from 'react';
import { loadStripe } from '@stripe/stripe-js';
import { Elements } from '@stripe/react-stripe-js';

// Aardvark keeps the resolved light/dark scheme in <html data-mantine-color-scheme> (see
the
// theme's color-scheme.js), updating it live when the reader toggles. Mirror it into React
so the
// Stripe Appearance below re-themes with the page.
function useColorScheme() {
  const read = () =>
    (typeof document !== 'undefined' &&
      document.documentElement.getAttribute('data-mantine-color-scheme')) ||
    'light';
  const [scheme, setScheme] = useState(read);
  useEffect(() => {
    const sync = () => setScheme(read());
    const obs = new MutationObserver(sync);
    obs.observe(document.documentElement, {
      attributes: true,
      attributeFilter: ['data-mantine-color-scheme'],
    });
    sync();
    return () => obs.disconnect();
  }, []);
  return scheme;
}

// The one thing a component library can't hand you from Markdown is the runtime Stripe
instance:
// loadStripe() returns a Promise (and injects Stripe.js), not a JSON-serialisable prop. So
this
// thin snippet owns the <Elements> provider; everything *inside* it — a Payment Element, a
Card
// Element — is rendered straight from the theme-declared `stripe` library via
// {% component('stripe', 'PaymentElement') %} etc., nested as this provider's children.
//
// loadStripe only runs in the browser, so it's resolved client-side; during the build-time
// prerender `stripe` is null and <Elements> just renders the children, then the real
instance
// mounts on hydration. The provider also carries the Appearance, switched with the page's
// light/dark scheme (the Payment Element honors Stripe's Appearance API), so it stays
readable.
//
// `deferred` (default true) drives the Payment Element's deferred intent
(mode/amount/currency).
// Pass `deferred={false}` for a plain card-input group (a CardElement, which can't live in
a
// deferred-mode Elements) — as the data-driven shop grid does, one provider per product.
```

```
export default function StripeProvider({
  publishableKey,
  amount = 2000,
  currency = 'usd',
  deferred = true,
  children,
}) {
  const stripe = useMemo(
    () => (publishableKey && typeof window !== 'undefined' ? loadStripe(publishableKey) :
null),
    [publishableKey],
  );
  const scheme = useColorScheme();
  const options = useMemo(() => {
    // Keep the init-only intent fields (deferred mode/amount/currency) separate from the
dynamic
    // appearance, so it reads clearly that only `appearance` changes on a theme toggle.
(Functionally
    // the same either way: react-stripe-js diffs the options and forwards only the changed
keys to
    // elements.update() — on a toggle that's just { appearance }.)
    const base = deferred ? { mode: 'payment', amount, currency } : {};
    return { ...base, appearance: { theme: scheme === 'dark' ? 'night' : 'stripe' } };
  }, [amount, currency, scheme, deferred]);
  return (
    <Elements stripe={stripe} options={options}>
      {children}
    </Elements>
  );
}
```

Composing it: a snippet is a tag

Defining a snippet gives you a `{% StripeProvider %}` tag, the same way a custom component does — so you compose it like any other directive, no `component()` call in sight. (`StripeProvider` has to be a snippet, not an `.md` component — it needs real React; see snippet vs. custom component.) The body of the tag becomes the provider's `children`; nest the built-in `{% card %}` and Stripe's `PaymentElement` (from the theme's `stripe` library) right inside it:

```
{% StripeProvider publishableKey="pk_test_…" %}
  {% card variant="plain" maw=480 cta="Pay $20.00" %}
    {% component('stripe', 'PaymentElement') %}
  {% endCard %}
{% endStripeProvider %}
```

Three sources, composed by nesting: your **`StripeProvider`** snippet owns the `<Elements>` context, the built-in **`{% card %}`** gives the frame and its `cta` button, and Stripe's **`PaymentElement`** — addressed through the theme's **`stripe`** library, since a theme library is always reached by its key — is the card's content. A snippet stays callable as `{% component('StripeProvider', …) %}` too;

you need that form inside a loop, as the storefront does below. Renders, live (test mode — swap in your own `pk_test_…`):

The Payment Element is a genuine Stripe iframe, themed to match the page — try toggling dark mode, or type the test card `4242 4242 4242 4242`. Nothing is charged: it's a publishable **test** key and there's no backend wired up here.

A storefront from a data file

Here's the payoff. Aardvark pages run real Python, and `data/` files are in scope as `data.<file>`. Point a loop at one and the library becomes a UI generator. This site ships `data/products.yaml`:

```
# data/products.yaml
items:
  - { name: Sticker Pack, price: 8,  icon: sticker-2, color: "#e8590c", blurb: "A dozen
die-cut vinyl aardvarks." }
  - { name: Enamel Mug,   price: 18, icon: mug,       color: "#1971c2", blurb: "12oz,
dishwasher-safe." }
  - { name: Zip Hoodie,   price: 55, icon: shirt,     color: "#7048e8", blurb: "Heavyweight
fleece." }
```

A `{% card %}` tag is expanded as the page is scanned, so it can't be written inside a `for` loop. The same built-in is reachable as `{% component('aardvark', 'card', …) %}` — the call form works anywhere, the loop included — and so is your snippet, as `{% component('StripeProvider', …) %}`. Loop over `data.products.items`, drop a real Stripe `CardElement` straight into each card body (one provider per card, since two `CardElement`s can't share an `<Elements>`), and lay them out in a `{% component('aardvark', 'cardGrid', …) %}`:

```
{%
cards = ''
for p in data.products.items:
    field = component('StripeProvider', publishableKey='pk_test_…', deferred=False,
        children=component('stripe', 'CardElement'))
    cards += component('aardvark', 'card', variant='plain', icon=p.icon, iconColor=p.color,
        title=p.name, subtitle=p.blurb, badge='$' + str(p.price), cta='Buy now',
children=field)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 2, 'lg': 3},
children=cards))
%}
```

renders, live:

Sticker Pack

[$8]

Enamel Mug

[$18]

Zip Hoodie

[$55]

A data file, the built-in card, and Stripe — composed in a short loop. Add a row to the YAML and a fourth card appears. That's the point of opening `{% component() %}` to any React library: your content and your data drive a real, interactive UI.

Options

Each library entry accepts a few extras beyond `package` and `components`:

- **`import`** — how the package exposes its components: `named` (the default — `import { X }`), `default` (a single default export; name it with `defaultExport`), or `namespace` (`import * as Lib`, which pulls the whole package in, so reach for it only when you must).
- **`css`** — a list of stylesheet imports the library needs (e.g. a design system's base CSS). They load with the rest of the island styles, on the client.
- **`ssr: false`** — skip this library during the build-time prerender. Use it for a browser-only library (one that touches `window` at import time); its islands then render on the client only.

If a declared package isn't installed, the build prints a warning and that library's components render as a harmless comment rather than failing the whole build — run the matching `npm install`.

CSS Selectors

A library component mounts inside an island wrapper that carries **both** the library key, in `data-aardvark-island`, so you can target one library's component without colliding with a same-named one elsewhere. The component itself renders its own markup (Stripe's Elements, for instance, mount a cross-origin iframe rather than Mantine-classed nodes).

```
[data-aardvark-island="PaymentElement"]                       /* every PaymentElement */
[data-aardvark-lib="stripe"][data-aardvark-island="PaymentElement"]  /* only the stripe
library's */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered element — pass it in the `component()` call form. (No standalone preview here: a `PaymentElement` only renders inside a Stripe `Elements` provider.)

Source: Markdown

```
{% component('stripe', 'PaymentElement', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''}) %}
```

Source: Python

```
component('stripe', 'PaymentElement', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

Examples

Two **built-in** tags for API documentation: `{% requestExample %}` and `{% responseExample %}`. Each wraps one or more **fenced code blocks** and renders a titled panel — a **Request** or a **Response** — with per-block **syntax highlighting** and **copy / download**. They're built to sit **pinned in a page's right-hand column** next to your prose, so a reader sees the call and the payload side by side; drop several blocks in one tag and they stack into a **tab strip** (curl / Python / JavaScript, or one panel per status code).

Both tags share one island, so everything below applies to both — only the default heading, icon, and accent color change (request vs. response).

Use them as `{% requestExample %}` / `{% responseExample %}` in Markdown, or call them from Python logic (loops, snippets) via `component('aardvark', 'requestExample', …)` / `component('aardvark', 'responseExample', …)`.

Usage

Wrap each example in a normal Markdown **code fence**. The fence's language drives the highlighting; everything between the tags is taken **verbatim** (so `{% … %}` inside the code is shown literally, never executed):

````
{% requestExample %}
```bash
curl https://api.example.com/v1/pets \
  -H "Authorization: Bearer $TOKEN"
```
{% endRequestExample %}
````

renders, live:

Request

bash

```bash
curl https://api.example.com/v1/pets \
  -H "Authorization: Bearer $TOKEN"
```

The response twin is identical, with a **Response** heading and a distinct accent:

Response

json

```json
{
  "id": 1,
  "name": "Rex",
  "species": "dog",
  "adopted": false
}
```

Multiple languages

Put **several fenced blocks** in one `{% requestExample %}` and the panel adds a **tab strip** — one tab per block. The tab label is the block's `title=` if it has one, otherwise its language. Perfect for showing the same call in curl, Python, and JavaScript:

````
{% requestExample %}
```bash
curl https://api.example.com/v1/pets
```
```python
import requests
requests.get("https://api.example.com/v1/pets")
```
```javascript
await fetch("https://api.example.com/v1/pets");
```
{% endRequestExample %}
````

renders, live:

Request

bash

```bash
curl https://api.example.com/v1/pets
```

python

```python
import requests
requests.get("https://api.example.com/v1/pets")
```

javascript

```javascript
await fetch("https://api.example.com/v1/pets");
```

Titling the panel and each block

Give the whole panel a heading with `title=`, and label an individual block with a `title="…"` **fence decorator** (the same one Markdown code fences accept). Block titles become the tab labels when there's more than one:

````
{% responseExample title="Responses" %}
```json title="200 OK"
{ "id": 1, "name": "Rex" }
```
```json title="404 Not Found"
{ "error": "pet not found" }
```
{% endResponseExample %}
````

renders, live:

Responses

200 OK

```
{ "id": 1, "name": "Rex" }
```

404 Not Found

```
{ "error": "pet not found" }
```

Pinning beside your prose

The panels are plain block islands, so they lay out wherever you place them. To get the classic **two-column API layout** — prose on the left, a sticky example panel on the right — drop the tag inside a Grid column (or your theme's right-rail). On a narrow screen the panel simply stacks under the prose.

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered panel root, the same on both tags:

Create a pet

bash

```
curl https://api.example.com/v1/pets -X POST \
  -H "Authorization: Bearer $TOKEN"
```

Source: Markdown

```
{% requestExample title="Create a pet" attr={'id': 'create-pet', 'data-analytics':
'api-example'} %}
```bash
curl https://api.example.com/v1/pets -X POST \
  -H "Authorization: Bearer $TOKEN"
```
{% endRequestExample %}
```

Source: Python

```
component('aardvark', 'requestExample', title='Create a pet',
          children='```bash\ncurl https://api.example.com/v1/pets -X POST\n```',
          attr={'id': 'create-pet', 'data-analytics': 'api-example'})
```

Options

Attribute	Applies to	Description
`title`	the tag	Overrides the panel heading (default **Request** / **Response**).
`title="…"`	a fence	A per-block heading; also the tab label when there are several blocks.
(fence language)	a fence	Drives syntax highlighting and the download file extension (`json`, `bash`, `python`, ...).
`attr`	the tag	Raw HTML attributes (`{…}`) forwarded onto the rendered panel root element (see above).

Syntax highlighting is baked in at **build time** (the same Pygments pass that colors every code fence on the site), so the reader's browser loads no highlighter for these panels; when a fence has no language — or site syntax highlighting is off — the block still renders as plain, copyable text.

Gitfolder

A **built-in** tag that embeds a small IDE-style browser for a folder in a **public** GitHub repo: a file tree on the left, the selected file on the right — **syntax-highlighted** source for code, **inline display** for images, and a **Source ⟷ Preview** toggle for Markdown and SVG — plus per-file **copy** / **download**, a link to each file's **source on GitHub**, and a **Download ZIP** of the **whole repository**. The repo's **license file** is linked at the bottom, so readers know the terms before reusing the code. It's perfect for showing an example project inline without copy-pasting every file into the page.

The files are fetched **at build time** — one archive download of the repo, from which only the requested subfolder is kept; never a `git clone` — and then **cached**, so a rebuild never re-downloads a folder it already has. **Browsing** loads nothing from GitHub: the file tree, the source and its highlighting, the rendered previews, and any displayed images are all baked into the page or served from your own site, so the widget works offline and stays fast. The one exception is the **Download ZIP** button, which links to GitHub's own archive URL for the repo — clicking it does reach GitHub, and it gives the reader the whole repository rather than just the folder shown.

Usage

Use a self-contained `{% gitfolder … %}{% endGitfolder %}` with a `github` (`owner/repo`):

```
{% gitfolder github="aardvarkdocs/sample-site" %}{% endGitfolder %}
```

renders, live:

Click a file in the tree to view it on the right; folders collapse. Code is syntax-highlighted, images show inline, and Markdown / SVG get a **Source ⟷ Preview** toggle — all with per-file copy / download and a link to the **source on GitHub**.

Both forms fetch the same thing — **one archive download** of the repo — so the choice is about the page, not the network. With no `folder`, every file in the repo is embedded (up to `maxFiles`, default 300), which makes a very large repo a heavy page. Scope to a `folder` (below) to embed just that subtree: a lighter page, and the `maxFiles` cap applies to the subtree rather than the whole repo.

Scope to a folder

Add a `folder` (a path within the repo) to show just that subtree:

```
{% gitfolder github="aardvarkdocs/sample-site" folder="content/getting-started" %}{%
endGitfolder %}
```

Folder mode downloads the same archive. A `folder=` browse fetches the repo's single zip archive and keeps only that subtree — the same one request the whole-repo form makes, to a host with no REST quota. **No `GITHUB_TOKEN` is needed** for either form, on CI or anywhere else. What `folder=` changes is the page: only that subtree's files are embedded, and the `maxFiles` cap applies to the subtree. The whole repo is still transferred to your build machine either way, so a `folder=` on a very large repo is a lighter page but not a lighter download.

Downloads & licensing

The **Download ZIP** button **always** fetches the **whole repository** from GitHub's own archive — even when you're browsing a single subfolder. That's deliberate: a zip of just part of a repo could leave out the `LICENSE`, and someone might reuse the code without realizing the terms. For the same reason the widget's footer points at the repo's license file, so the terms are one click from the code they govern.

The footer quotes your license file; it does not classify it. Aardvark reads the root-level license file out of the same downloaded archive that provides the file previews, takes the **title line the file gives itself**, and shows that, linked — no GitHub API, so nothing to authenticate and nothing to be rate-limited. It never decides which license you are under, so it can never name the wrong one and there is no list of known licenses to fall off: a brand-new SPDX id or your own bespoke terms appear exactly as your file words them.

The footer has three things it can say:

- **the title, linked** — `MIT License`, `GNU AFFERO GENERAL PUBLIC LICENSE`, `The Acme Public License v3`, whatever your first line says (Markdown heading marks and underlines come off; the words don't).
- **`License`, linked** — the file is there but doesn't name itself: it opens with a copyright notice (BSD, ISC and many MIT copies do), or it's a symlink, a binary, or too large to read. You still get the link, which is the part that matters.
- **no license detected — assume all rights reserved** — the repo root has no license file. This is only ever said about a repo that was actually read: a fetch that failed has no files either, so that widget shows its usual "couldn't fetch this" fallback instead of a footer.
- **no footer at all** — nothing read this repo's root, so there is nothing honest to say. You see this in one narrow case: you upgraded Aardvark, the cached copy of the repo was written by the older version, and this build couldn't reach GitHub to re-read the license. The files are still shown (they are exactly what the previous build stored) and the build warns; the next build that reaches GitHub settles the license and the footer comes back.

Only the best-named root file is read — `LICENSE`, then `LICENCE`, `COPYING`, `UNLICENSE`, then any name that carries one of those words (`LICENSE-MIT`, `MIT-LICENSE`, `COPYING-LGPL` — the generic word decides, never the qualifier). A repo carrying several is quoting itself, not disagreeing with itself, so the link goes to the best-ranked one. A bare `COPYRIGHT` is **not** one of them: it states who owns the code, not what you may do with it, so a repo whose root holds only that gets the all-rights-reserved footer rather than a link labelled `License`.

The license you see is a cached snapshot. The footer's license — like the file previews — comes from the fetch that filled the cache, so it describes the code you actually downloaded. The **Download ZIP** button is different: it's a live link to GitHub's archive, so it always delivers the repo's current contents (or your pinned `ref=`). On a fresh build the two agree; on a stale local cache the shown license can lag what the ZIP now contains, until you refresh the cache or set `cache="false"`. In practice your **published** site is always current (every deploy re-fetches — see Caching), so this is only ever a local-preview nuance. To lock the whole widget — previews, license, and ZIP — to

one immutable version, pin a `ref=` (a tag or commit); the license is then resolved for that exact ref.

How each file is shown

- **Code & text** — syntax-highlighted exactly like a fenced code block, with the same copy / download actions.
- **Images** (`png`, `jpg`, `gif`, `webp`, `svg`, …) — displayed inline. The image is served from your own site (baked in at build time), so the page stays self-contained.
- **Markdown & SVG** — get a **Source ⟷ Preview** segmented control: Markdown toggles between its highlighted source and the rendered document; SVG toggles between its markup and the rendered image. (In a Markdown preview, relative image and link paths are rewritten so they resolve — to your served copy when the image is in the folder, otherwise to GitHub.)
- **Other binaries & very large files** — not previewed inline; grab them from the **Download ZIP** button, which always contains the **complete repository** (see below).

Caching

By default (`cache="true"`) the folder is downloaded **once** and stored under `.aardvark-cache/` (which is git-ignored). Every later build reuses those files and touches the network only if the cache is missing — so builds stay fast and work offline once primed. **Delete the cache** (or that folder within it) to pull a fresh copy.

Caching is mostly a local convenience. Because `.aardvark-cache/` is git-ignored it's never committed, so a deploy host that builds from a fresh checkout starts with an empty cache and re-downloads on every deploy — which means your **published** site always reflects the latest fetch, and `cache="true"` vs `cache="false"` makes little practical difference there. Locally, the cache is what keeps repeat builds fast and offline-friendly. (If your CI persists build directories between runs, the cache can carry over there too — clear it to force a refresh.)

A warm cache doesn't hide files being too large. If files were skipped for size, or the folder was cut by `maxFiles`, every build says so — not just the one that downloaded. You never have to clear the cache to find out what the widget isn't previewing.

One question a warm build can't always answer: if you lower `maxFileBytes` on a folder that was itself cut by `maxFiles`, the files past that cut aren't on disk to re-check. The build says which figures it can't give you rather than quoting one it can't stand behind — delete the cache for exact numbers.

Two notices only a downloading build can give, because they describe archive entries that were never written to the cache: a file skipped for an unsafe path, and one skipped as corrupt or over-large when unpacked. A warm build renders exactly what a cold one would, but stays silent about both. If you're chasing either, delete `.aardvark-cache/gitfolder` and rebuild.

A warm build shows the same files a fresh download would — including after you lower `maxFiles`. One shape of cache can't promise that, and says so instead of guessing: the message tells you the

list may differ and leaves the call to you. One re-download retires it for good.

Upgrading Aardvark re-downloads once, and only for the license. A cache written before Aardvark read licenses out of the archive carries an answer from GitHub's old API — including "no license", which that API also recorded when it simply couldn't reach GitHub. This version repeats neither: the first build after the upgrade fetches that repo once, reads the license out of the archive, and stays warm from then on. If that one build can't reach GitHub, your cached files still render — the failure costs you the footer, not the widget: Aardvark warns, shows the files it already has, and renders no license line at all, because it has no honest answer to put there. The next successful build settles it.

Nothing else re-downloads on an upgrade: a folder cached by an older version may not be able to answer the questions above, and the build says so and leaves your cached files alone rather than silently replacing them with whatever the branch points at today. Delete `.aardvark-cache/gitfolder` to refresh it.

Besides that one-time license migration, what re-downloads is **changing a limit**, and not every change:

- **Raising `maxFiles` or `maxFileBytes` re-downloads.** You've asked for more than the snapshot holds. (A cache old enough not to record the limits it was fetched under can't hear the ask and stays warm; delete it once, and every build after that answers normally.)
- **Lowering `maxFileBytes` re-downloads only when the lower cap leaves the snapshot short.** If enough cached files still fit under it, the stored copy already is what a fresh download would show, and nothing is fetched. Those builds do re-check the whole cached folder, though, to work out which files the lower cap now hides — a cost a big folder keeps paying on every build until something re-fetches it.
- **Lowering `maxFiles` alone never re-downloads.** It shows fewer of the files already cached.

A re-download on an unpinned `ref` brings today's contents. Pin a `ref=` if that matters, or delete the cache first so the refresh is one you chose.

A half-written cache is re-downloaded, not served. If a build was killed mid-save, or the cache was half-deleted by hand, there's no telling how much of the folder is missing — so the build fetches it again rather than quietly rendering whatever happens to be on disk. (A build that is currently saving that folder is recognized and passes without complaint.)

Two builds can share a project safely. A `vark dev` running beside a `vark build` — or two dev servers, or CI beside a shell — can't corrupt each other's cached folders: the one that gets there first saves the folder and the other leaves it alone, mentioning that it did. Nothing waits, so an overlap costs one extra download rather than a stalled page.

The one case worth knowing about: a build **killed while saving** leaves a lock file behind at `.aardvark-cache/gitfolder/<key>.json.lock`, and nothing ever removes a lock it doesn't own. Nothing new can be cached for that folder until the file goes. Whether that means re-downloading depends on what the interrupted build had already done: a snapshot it had finished publishing keeps being served, but one cut off mid-save took its sidecar with it, and that folder does re-download on every build. After a few minutes the build warns and names the exact path. Check

that no `vark` process is still running, then delete it — with nothing running that's safe, and the next build caches normally.

Set `cache="false"` to re-download on **every** build — you always get the latest, at the cost of a network round-trip (and longer builds) each time:

```
{% gitfolder github="aardvarkdocs/sample-site" folder="content/getting-started"
cache="false" %}{% endGitfolder %}
```

With `cache="false"`, the build prints a note each time (`Aardvark: note: …`) saying that it's re-fetching and **how long** the download took, so a slow build is never a mystery. It's a status line, not a warning — re-fetching is what you asked for — so it isn't counted in the build summary's warning total.

Options

Attribute	Effect
`github="owner/repo"`	The **public** repo (required). A full `https://github.com/owner/repo` URL or a trailing `.git` is tolerated.
`folder="path/in/repo"`	The folder to show, relative to the repo root. **Optional** — omit it to browse the whole repo root. A leading `/` is fine.
`cache="false"`	Re-download every build (ephemeral, always latest), with a timed note in the build output. Default `true`: download once, then reuse the local cache.
`ref="…"` / `branch="…"`	Pin a branch, tag, or commit SHA. Default: the repo's **default branch**.
`label="…"`	Accessible name for the widget (sets `aria-label`); useful when a page has more than one.
`height="480"`	Override the panel height (pixels, or any CSS length).

Notes

- **Public repos only.** Private repos aren't supported (and would need credentials the build doesn't have).
- **The widget is wide.** It works at the default content width, but a file browser has more room to breathe on a `mode: wide` (or `mode: full`) page — set that in the page's front-matter.
- **No API rate limit to worry about — at all.** Everything the widget shows comes from one archive download, which has no REST quota: the file content and the footer's license, which is detected locally from the archive's own license file. The widget makes **no GitHub API call**, so no

`GITHUB_TOKEN` is needed on any host — including shared-IP CI (Cloudflare Pages, Netlify, GitHub Actions), where the anonymous 60/hour API ceiling is shared across every project on the runner and used to cost the footer its license line. There is no credential to configure and nothing for someone else's builds to exhaust. (The old `gitfolder.token` config key is retired; a leftover one warns so you remember to remove it.)

- **Resilient by design.** If a repo or folder can't be fetched (offline build, a typo, a private or deleted repo, a network failure), the build prints a warning and the widget renders nothing — it never fails the whole build.
- **Turn it off site-wide** with `gitfolder: false` in `aardvark.config.yaml`.

CSS Selectors

The browser mounts inside an island wrapper carrying `data-aardvark-island="GitFolder"` and renders its own class names — target the panel, the header, the file tree, and each row.

```
[data-aardvark-island="GitFolder"]  /* the island wrapper */
.aardvark-gitfolder                 /* the panel */
.aardvark-gitfolder-header          /* the title / toolbar row */
.aardvark-gitfolder-tree            /* the file list */
.aardvark-gitfolder-row             /* a single file/folder row */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered file-browser root. (Style it through the CSS parts above, and configure it with the documented attributes: `github`, `folder`, `ref`, `cache`, `label`, `height`.)

Source: Markdown

```
{% gitfolder github="aardvarkdocs/sample-site" folder="content/getting-started"
attr={'data-analytics': 'repo-tree', 'aria-label': 'Repository files'} %}{% endGitfolder %}
```

Source: Python

```
component('aardvark', 'gitfolder', github='aardvarkdocs/sample-site',
          folder='content/getting-started',
          attr={'data-analytics': 'repo-tree', 'aria-label': 'Repository files'})
```

Include

A **built-in** tag that renders another file in place:

```
{% include '/path/to/file.md' %}
```

The included file is rendered through the **same `{% %}` engine and the same page namespace**, so a partial can read the including page's front matter (`page.*`) — which may hold arbitrary data — and use ordinary conditional logic to vary what it emits. The result is then parsed as part of the page's single Markdown pass, so a `.md` partial's headings, lists, and components all render normally.

Where partials live

A file whose name (or any parent directory) starts with an underscore — e.g. `content/_partials/note.md` — is a **partial**: it's available to `{% include %}` but is never published as its own page. Drop reusable fragments under a `_partials/` directory and include them anywhere.

Path resolution

Path form	Resolves against	Example
Leading `/`	the content root (per language)	`{% include '/_partials/note.md' %}` → `content/_partials/note.md`
No leading `/`	the **including file's** directory	`{% include 'note.md' %}` → a sibling of the current page

The argument is a Python expression, so the path can be computed — a variable (`{% include page.partial %}`) or a conditional (`{% include '/_partials/pro.md' if page.edition == 'pro' else '/_partials/free.md' %}`). A missing file or an include cycle fails the build with a clear error.

Example

This partial lives at `content/components/_partials/plan-note.md`. It reads two front-matter fields off whatever page includes it and branches on them:

```
{%
edition = page.get("edition", "free")  # the "edition" field, or "free" if the page omits it
product = page.get("product", "this product")  # the "product" field, or a generic fallback
if edition == "pro":
    plan = "Pro"
    note = "Every feature here is included."
else:
    plan = "Free"
```

```
    note = "Some features below require an upgrade."
%}
> **{% product %} — {% plan %} plan.** {% note %}
```

Including it:

```
{% include '/components/_partials/plan-note.md' %}
```

renders, live:

> **Aardvark Cloud — Pro plan.** Every feature here is included.

This page's front matter sets `product: "Aardvark Cloud"` and `edition: pro`, so the partial emits the **Pro** line above. A page that set `edition: free` — or omitted it — would get the other line from the same partial, with no change to the partial itself. That's the point: one shared fragment, varied per page by the page's own data.

CSS Selectors

`{% include %}` adds no wrapper of its own — it splices the partial's rendered Markdown in place, so the result is exactly the elements the partial emits. Style those with their own selectors (whatever headings, lists, or component classes the partial produces); there is no `include` element to target.

Injecting Attributes

Because `{% include %}` renders no element of its own, it has no `attr={…}` channel. To vary what a partial emits per page, branch on the including page's front matter (`page.*`) inside the partial, as shown above.

Map

A **built-in** tag that embeds an interactive map with a marker for each location. It renders OpenFreeMap vector tiles with MapLibre GL JS — both **free and keyless** — so a map just works with nothing to sign up for.

Aardvark uses the audited `mantine-map` package as an internal implementation dependency; it does not expose a second community tag. `{% map %}` is the only map authoring surface. The MapLibre browser runtime and stylesheet are bundled from the exact version installed in your project, and Aardvark publishes its content-hashed worker locally. The executable map engine therefore loads from your own site rather than a JavaScript CDN. The selected remote style, tiles, glyphs, and sprites are provider data, not executable CDN runtime.

Drop a pin by **street address** and the address is turned into coordinates once, at build time, by the free Nominatim (OpenStreetMap) geocoder — cached so a rebuild never re-looks-up an address it already knows. Prefer to be exact (or build offline)? Give a pin explicit `lat` / `lng` and no geocoding happens at all.

Usage

Wrap the map in `{% map %} … {% endMap %}` and give each location its own self-closing `{% pin %}`. The simplest form is an address and a label:

```
{% map %}
{% pin address="British Museum, London" label="British Museum" %}
{% pin address="Tower of London, London" label="Tower of London" %}
{% pin address="Buckingham Palace, London" label="Buckingham Palace" %}
{% endMap %}
```

At build time each `address` is geocoded and baked into the page as coordinates, so the reader's browser only loads the basemap and drops the markers — it never geocodes.

Live example

The same three London landmarks, pinned by explicit coordinates so the page needs no network when it builds — the source below and the map it renders match exactly. With no `center` or `zoom` set, the map frames all the pins automatically:

```
{% map height=420 %}
{% pin lat=51.5194 lng=-0.1270 label="British Museum" description="Great Russell St" %}
{% pin lat=51.5081 lng=-0.0759 label="Tower of London" description="A royal fortress since
1066" %}
{% pin lat=51.5014 lng=-0.1419 label="Buckingham Palace" color="#c2255c" %}
{% endMap %}
```

renders, live:

[Interactive map — view it online at https://aardvarkdocs.com/components/extras/map/]

Click a marker for its popup. The map keeps **MapLibre | OpenFreeMap © OpenMapTiles | © OpenStreetMap contributors** attribution in the corner — that crediting is required, so don't remove it.

Pinning a location

A `{% pin %}` is placed one of two ways:

- **By address** — `{% pin address="350 Fifth Ave, New York, NY" %}`. Geocoded at build time (cached). Easiest to author; needs a network connection the first time it's built.
- **By coordinates** — `{% pin lat=40.7484 lng=-73.9857 %}`. Exact, reproducible, and never touches the network. Use this for precision, for places a geocoder won't find, or to keep a build fully offline.

An address that can't be resolved (typo, or an offline build with a cold cache) is **skipped with a build warning** rather than failing the build — so one bad address never breaks your docs.

Options

Every `{% map %}` attribute is optional:

`{% map %}` attribute	Effect
`style="liberty"`	Basemap style: `liberty` (default), `bright`, `positron`, or a full MapLibre style URL.
`zoom=12`	Initial zoom level. Omit to auto-fit the pins.
`center="51.50,-0.12"`	Center as `"lat,lng"` — or an address. Omit to auto-fit the pins.
`height=420`	Map height in pixels (default `400`).
`interactive=false`	Lock the map (no pan/zoom, no zoom buttons) for a static locator.

Each `{% pin %}` takes:

`{% pin %}` attribute	Effect
`address="…"`	Street address, geocoded at build time. Use this **or** `lat`/`lng`.
`lat=… lng=…`	Explicit coordinates (skips geocoding). `lon` is accepted as an alias for `lng`.

`label="…"`	Bold heading in the marker's popup (and the location's name in the fallback list).
`description="…"`	A line of detail below the label in the popup.
`color="#c2255c"`	Marker color (any CSS color).

Geocoding, privacy & offline builds

Geocoding runs **only at build time** and only for `address` pins — the published page contains coordinates, never an address lookup. Results are cached under `.aardvark-cache/` (git-ignored), so addresses are resolved once and reused on every later build.

Forcing a re-lookup

Each geocoded address is cached as one small JSON file under `.aardvark-cache/geo/`, named by a hash of the address (its contents are the resolved `lat` / `lng` and place name). The cache is purely derived and git-ignored, so clearing it is always safe — it just costs a fresh lookup on the next build. To drop **every** cached coordinate and re-resolve on the next build:

```
rm -rf .aardvark-cache/geo
```

To drop a **single** entry — say the geocoder placed a pin in the wrong spot and you want it looked up again — the filenames are hashed, so match the file by the (wrong) coordinate you saw on the map, then delete it and rebuild:

```
grep -rl 43.6532 .aardvark-cache/geo | xargs rm
```

(To skip geocoding for a location altogether, give its pin explicit `lat` / `lng`.)

The default geocoder is Nominatim's public server, used within its usage policy: one request per second, a descriptive `User-Agent`, and aggressive caching. For heavier use, switch to Google — all via the optional `map` block in `aardvark.config.yaml`:

```
map:
  geocoder: nominatim          # default; or "google"
  # googleApiKey: "…"          # required when geocoder: google
  # rateLimit: 1.0             # requests per second
  # timeout: 10                # seconds per geocoder request
  style: liberty               # default basemap
```

`mantine-map` and MapLibre GL JS are exact dependencies in a scaffolded site's `package.json`: the former is the audited internal adapter, while the direct MapLibre pin keeps worker resolution on the same installed tree. During the islands build Aardvark bundles that runtime and CSS and emits a self-contained, content-hashed worker under `/_aardvark/maplibre/`. There is no runtime JavaScript or CSS CDN dependency to configure.

Content-Security-Policy: allow the local worker with `worker-src 'self'`, and allow MapLibre's image forms with `img-src 'self' data: blob:`. The basemap style still names its own remote style, tile, glyph, sprite, and image providers, so add those origins under `connect-src` and `img-src` as appropriate. These responses are map data, not executable runtime fetched from a CDN. The default OpenFreeMap styles use OpenFreeMap-hosted resources. (Cloud hosts and WAFs sometimes set a CSP for you — check there if a map renders locally but loses its basemap after deployment.)

No JavaScript? A reader (or a search crawler) without the map still gets an accessible list of the pinned locations, each linking to its spot on OpenStreetMap — the same list screen readers use.

CSS Selectors

The map mounts inside an island wrapper carrying `data-aardvark-island="Map"` and renders its own class names — target the container, the MapLibre canvas, and the no-JavaScript fallback list.

```
[data-aardvark-island="Map"]  /* the island wrapper */
.aardvark-map                 /* the map container */
.aardvark-map-canvas          /* the MapLibre canvas */
.aardvark-map-fallback        /* the no-JS location list */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered map root. (Style it through the CSS parts above, and configure the map with the documented attributes — `style`, `zoom`, `center`, `height`, `interactive` — plus each `{% pin %}`.)

[Interactive map — view it online at https://aardvarkdocs.com/components/extras/map/]

Source: Markdown

```
{% map height=420 attr={'data-analytics': 'office-map', 'aria-label': 'London landmarks'} %}
{% pin lat=51.5194 lng=-0.1270 label="British Museum" %}
{% pin lat=51.5081 lng=-0.0759 label="Tower of London" %}
{% pin lat=51.5014 lng=-0.1419 label="Buckingham Palace" %}
{% endMap %}
```

Source: Python

```
component('aardvark', 'map', height=420,
          attr={'data-analytics': 'office-map', 'aria-label': 'London landmarks'},
children='''
{% pin lat=51.5194 lng=-0.1270 label="British Museum" %}
{% pin lat=51.5081 lng=-0.0759 label="Tower of London" %}
{% pin lat=51.5014 lng=-0.1419 label="Buckingham Palace" %}
''')
```

OpenAPI

A **built-in** tag that renders an OpenAPI spec as a Mantine-powered reference — parameter and response tables, multi-language request samples, and a **"Try it now"** form — **inline, on any page**. Point it at a **single operation** to splice one endpoint into a guide right where you describe it; point it at the whole spec for a complete reference.

This page is about that **single-endpoint slice** — dropping one operation into the flow of a page. For the whole spec rendered as a full, immersive reference, head to this site's **Gateway API** tab — the same tag with no slice, on a `mode: wide` page, rendering this site's own live gateway API.

Usage

Put your spec under `openapi/` and pass **both** `verb` and `endpoint` to render just that one operation, inline. Paths are relative to the **site root** (where `aardvark.config.yaml` lives):

```
{% openapi 'openapi/petstore.json' verb='get' endpoint='/pet/{petId}' %}
```

The two attributes go together. Supply only one and the build stops with advice:

```
{% openapi %} got verb='get' but no endpoint. Rendering a single operation
needs BOTH verb and endpoint, e.g.
{% openapi 'openapi/petstore.json' verb='get' endpoint='/your/endpoint' %}.
Omit both to render the whole spec.
```

`endpoint` must match the spec's path exactly — leading slash and any `{param}` placeholders included. A bad pair lists the operations that do exist.

Here's a live slice — only `GET /pet/{petId}`, rendered right here in the middle of this page:

Swagger Petstore - OpenAPI 3.0

v1.0.27

Browse and manage pets 🐾, store orders, and users — see the overlay guide. This reference is the upstream **Swagger Petstore** spec, customized entirely through an aardvark overlay.

Sandbox API

Every operation here runs against the public Petstore sandbox — data resets periodically, so experiment freely.

🔑Authentication

Most write operations need an API key — paste one into the **Authorize** box and it rides along with every Try it now call.

❖Status codes

Errors follow standard HTTP semantics: `4xx` for client problems, `5xx` for ours.

Base URL: `https://petstore3.swagger.io/api/v3`

`GET /pet/{petId}`

Find pet by ID.

Returns a single pet.

Parameters

Name	In	Type	Required	Description
`petId`	path	integer (int64)	yes	ID of pet to return

Responses

Code	Description
`200`	successful operation
`400`	Invalid ID supplied
`404`	Pet not found
`default`	Unexpected error

That's the point of a slice: the endpoint's full reference — its parameters, responses, request samples, and **Try it now** form — sits inline next to your prose, instead of sending the reader off to a separate reference page. Its operation also joins this page's left-hand nav, the same as a heading.

The whole spec

Omit `verb` and `endpoint` and the directive renders **every** operation — that's the entire reference:

```
{% openapi 'openapi/petstore.json' %}
```

Because it's ordinary page content, the page chooses its own layout — give it `mode: wide` so the response tables get room to breathe. This site's **Gateway API** tab does exactly that — the same tag, no slice, on a wide page, with every operation wired into the left-hand nav. It renders this site's real Aardvark gateway API, so it doubles as a live reference and a demonstration.

Operations are grouped in the nav by their first OpenAPI `tag`. The bucket heading is the tag's Redoc-style `x-displayName` when set (else the first line of its `description`, else the sentence-cased tag name) — so a tag like `pet` reads as **Pets!** rather than a bare lowercase `pet` when both fields are present.

Responses

Operations list their responses straight from the spec. Define a `responses` block and Aardvark renders one collapsible row per status code — success and errors alike — colour-coded by class (green `2xx`, orange `4xx`, red `5xx`). Every row documents:

- the response **description**,
- any **headers** the response declares (name, type, description),
- a **properties table** for the body schema (name, type, required, description), resolving `$ref`s into `components/schemas` and descending into array items,
- the **example body**, shown in the right-hand samples rail (beside the request samples on a wide page) as an **interactive, collapsible JSON tree** — expand or collapse any node and copy a single value or the whole body.

```yaml
responses:
  '200':
    description: The requested pet.
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Pet'
        example:
          id: 7
          name: Fido
          status: available
  '404':
    description: No pet with that id exists.
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
        example:
          code: 404
          message: pet 7 not found
```

The interactive tree is the same community viewer behind `{% jsontree %}` (the `@gfazioli/mantine-json-tree` package). It's already bundled on this site; if your own project doesn't have that package installed, response examples render as a plain highlighted code block instead — run `npm install @gfazioli/mantine-json-tree` to upgrade them to the tree. Non-JSON examples always use the code block.

Authorization

Declare a security scheme and Aardvark renders an **Authorization** box at the top of the reference. Readers paste their API key once; by default it's held in memory for that browser session only, with a **Save this key in my browser** checkbox to keep it (in `localStorage`) for next time. Either way it flows into both the request samples and the **Try it now** requests, placed however the scheme declares it.

Add the scheme under `components.securitySchemes` and apply it — site-wide with a top-level `security`, or per operation:

```yaml
security:
  - bearerAuth: []
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: API key
      description: API key sent as a bearer token in the Authorization header.
```

Bearer tokens (`Authorization: Bearer <key>`), API keys in a header, query string, or cookie (`type: apiKey` with `in: header`/`query`/`cookie`), and HTTP Basic (`type: http, scheme: basic` — paste `username:password`, base64-encoded for you) are all handled — though cookie keys appear only in the request samples, since the in-browser Try it now can't set cookies. An operation opts out of auth with `security: []`; a spec that defines a single scheme but omits `security` still shows the box and applies that scheme.

Request samples

Every operation shows ready-to-run **request samples**, generated from the spec and switchable by tab — **cURL**, **Python** (`requests`), **JavaScript** (`fetch`), **Go** (`net/http`), **PHP** (curl), and **Rust** (`reqwest`).

Samples fill in the path and query parameters, include the example request body for writes, and carry whatever credential the security scheme declares — for a bearer scheme that's an `Authorization: Bearer YOUR_API_KEY` header (an `apiKey` scheme puts it in its declared header, query parameter, or cookie instead), shown as a placeholder until you enter a key. Anything you type into **Try it now** (parameters, body) is mirrored into the samples, so the snippet always matches the request you're about to send.

Rich descriptions

Every `description` in the spec — the API blurb, each operation, parameters, request-body fields, responses, headers, schema properties, and security schemes — renders as **Markdown**, so links, emphasis, lists, and code all work. Standalone descriptions (the API intro and each operation) go further: they accept the **full set of Aardvark primitives**, so you can drop a callout, a card grid, tabs, or any other component straight into your API docs:

```
# An operation's description:
Returns a single pet by id.

{% callout severity="caution" title="Rate limited" %}
This endpoint is capped at 60 requests per minute per key.
{% endCallout %}
```

Operation and API descriptions render as **block** content (they can hold block components like callouts and card grids). Inline descriptions — parameter, schema-property, and response-header rows in their tables, plus each **response's** own blurb beside its status badge — render as **inline** Markdown, so they take emphasis, code, links, and inline primitives like `{% icon %}`. These sit in tight spots (a table cell, an accordion label), so keep them short: a block primitive like a callout dropped in an inline slot warns at build time and belongs in an operation or API description instead. Summaries and titles stay plain text; they're also used as headings and nav labels.

Descriptions are processed just like page content — the template engine runs before Markdown — so wrap anything you want to show rather than run in a `raw` block, the same as anywhere else on a page.

Rendered description HTML is sanitized (descriptions can come from a spec or overlay you don't control), which drops the inline `style` attribute — so Markdown **table column alignment** (`|:--|--:|`) isn't preserved inside an API description, though the table itself still renders. Use a plain table, or a page body, if alignment matters.

Sanitizing strips scripts, event handlers, and `javascript:` URLs, but it allows images and links — including external ones, exactly like a Markdown image anywhere else. An external `<img>` still makes an outbound request when the page is viewed, so a spec or **overlay pulled from a third party you don't vet** could phone home (a tracking pixel). Treat an unvetted overlay like any other third-party content you'd embed.

Overlays

Often the spec you document isn't yours to edit — a vendor's file, or one generated by a build you don't control. An **OpenAPI Overlay** customizes a base spec without touching it: a side file of `actions`, each selecting node(s) with a JSONPath `target` and either **`update`**-ing or **`remove`**-ing them. Fix a wrong summary, hide an internal operation, enrich a parameter, or drop an Aardvark callout into the overview — all from a file that survives the next time you re-download the spec.

Setting one up

Aardvark discovers overlays **by filename** — no directive argument, no config. Drop a file beside the spec named `<spec-stem>.overlay.{yaml,yml,json}` and it's applied on every build. For `openapi/petstore.json` the stem is `petstore`, so the file is `openapi/petstore.overlay.yaml`. The shape is an `overlay:` version, an optional `info:` block (the overlay's own title/version — Aardvark doesn't merge it into your spec), and the `actions` list:

```
# openapi/petstore.overlay.yaml — sits beside petstore.json, applied automatically
overlay: 1.0.0
```

```
info: { title: Petstore docs overlay, version: 1.0.0 }
actions:
  - target: "$.paths['/pet'].post.summary"   # what to select (a JSONPath)
    update: "Create a new pet listing"        # what to do with it
```

Each action needs a `target` and **exactly one** of `update` or `remove` — both together, or neither, is skipped with a warning.

Targeting nodes

`target` is a JSONPath into the spec — the **extended** dialect, so filter expressions work. Common shapes:

Target	Selects
`$.info.description`	the API's overview blurb
`$.info.title`	the API title
`$.paths['/pet'].post.summary`	one operation's summary
`$.paths['/pet/{petId}'].get`	a whole operation (e.g. to `remove` it)
`$.paths['/pet/{petId}'].get.parameters`	that operation's parameter **list**
`$.paths['/pet/findByStatus'].get.parameters[?(@.name=='status')]`	one parameter, picked by name
`$.components.schemas.Pet.properties.status.description`	a schema field's description
`$..parameters[?(@.name=='petId')].description`	**every** `petId` parameter, anywhere in the spec
`$`	the whole document (root)

A `target` that matches nothing — a typo, or an operation that isn't there — emits a build warning and is skipped, so an overlay can't break the build.

`update` — rewrite, merge, or extend

What `update` does depends on the node it lands on:

- **A string (or other scalar) is replaced** — the summary action above swaps in new text.

A mapping deep-merges. Where both sides hold a mapping the two merge key-by-key; otherwise your value wins. (It's the Overlay merge, not JSON Merge Patch — a `null` **replaces**, it doesn't delete; use `remove` to delete.) Add or override a field without restating the whole object:

```
- target: "$.components.schemas.Pet.properties.status"  update: { description: "Lifecycle status.", example
```

A list extends. On an array, a **list** value appends each of its items; a single object or scalar is appended as one element. So you can add a parameter without rewriting the ones already there:

```
- target: "$.paths['/pet/findByStatus'].get.parameters"  update:    - { name: limit, in: query, schema: { t
```

- **The root `$` deep-merges into the whole document** (with a mapping value) — the way to add something top-level like a `servers` entry or a site-wide `security` requirement.

`remove` — hide a node

`remove: true` deletes the selected node — an internal operation, a deprecated parameter, a property you'd rather not show:

```
  - target: "$.paths['/store/inventory'].get"   # drop one operation from the reference
    remove: true
```

`remove` must be the literal boolean `true`: a quoted `"true"`, or `remove: false`, is ignored (with a warning) so a stray value can't silently delete half your spec. The document root `$` can't be removed.

Injecting Aardvark primitives

Because a spec's standalone descriptions render as **content** (see Rich descriptions), an `update` that targets a `description` can drop a **callout, card grid, icon — any primitive** straight into the rendered reference. This is the overlay's headline trick for docs: enrich a vendor's bare spec with the same components you use on a page, without forking it. (Wrap directives in the overlay's YAML exactly as on a page — the engine runs over description text too.)

Layering several overlays

Need more than one — say auth tweaks kept separate from prose? Name them with numeric prefixes and they apply in **lexical filename order**:

```
petstore.overlay.10-auth.yaml    # applied first
petstore.overlay.20-prose.yaml   # then this
petstore.overlay.yaml            # then this (a bare name sorts last)
```

The sort is lexical, not numeric — **zero-pad** single digits (`01`, `02`, … `10`), or an unpadded `2-` sorts after `10-`.

Best-effort by design

Overlays never fail a build. A file that won't parse, an action missing its `target`, an unsupported JSONPath, a `target` that matches nothing — each emits a **build warning** (it shows in the end-of-build summary) and is skipped; the rest of the overlay still applies. An `overlay:` version that isn't `1.x` warns but applies anyway. Treat an overlay you didn't write like any third-party content — see the sanitizing note under Rich descriptions.

Live demonstration

Every petstore reference on this page — the slices above and below — is overlaid by the real `openapi/petstore.overlay.yaml` sitting beside the spec, applied automatically. Four actions on the upstream Swagger Petstore spec (prose abridged here):

```
# openapi/petstore.overlay.yaml — sits beside petstore.json, applied automatically
overlay: 1.0.0
info: { title: Petstore docs overlay, version: 1.0.0 }
actions:
  # 1. Replace the API blurb with prose + block primitives (a callout and a two-card grid).
  - target: "$.info.description"
    update: |
      Browse and manage pets {% icon "paw" %}, store orders, and users.

      {% callout severity="info" title="Sandbox API" %}
      Every operation runs against the public Petstore sandbox — data resets periodically.
      {% endCallout %}

      {% cardGrid cols=2 %}
      {% card title="Authentication" icon="key" accent="grape" %}
      Most write operations need an API key.
      {% endCard %}
      {% card title="Status codes" icon="components" accent="grape" %}
      Errors follow standard HTTP semantics.
      {% endCard %}
      {% endCardGrid %}

  # 2. Rewrite a summary you can't fix upstream.
  - target: "$.paths['/pet'].post.summary"
    update: "Create a new pet listing"

  # 3. Drop a caution callout into one operation's description.
  - target: "$.paths['/pet/findByStatus'].get.description"
    update: |
      Returns every pet matching the given **status** values.

      {% callout severity="caution" title="Deprecation" %}
      The multi-value form is being phased out — prefer a single status per request.
      {% endCallout %}
```

```
  # 4. Enrich one parameter, picked by name with a filter expression.
  - target: "$.paths['/pet/findByStatus'].get.parameters[?(@.name=='status')].description"
    update: "Filter by status — one or more of `available`, `pending`, or `sold`."
```

All four are visible right here. **Action 1** rewrites the API blurb — it renders as the **overview** that heads every reference on this page (the **Sandbox API** callout and the two-card grid, swapped in for the bare upstream text). **Action 2** rewrites a summary; here's a live `POST /pet` slice showing **"Create a new pet listing"** instead of the upstream wording:

Swagger Petstore - OpenAPI 3.0

v1.0.27

Browse and manage pets 🐾, store orders, and users — see the overlay guide. This reference is the upstream **Swagger Petstore** spec, customized entirely through an aardvark overlay.

Sandbox API

Every operation here runs against the public Petstore sandbox — data resets periodically, so experiment freely.

🔑Authentication

Most write operations need an API key — paste one into the **Authorize** box and it rides along with every Try it now call.

❖Status codes

Errors follow standard HTTP semantics: `4xx` for client problems, `5xx` for ours.

Base URL: `https://petstore3.swagger.io/api/v3`

`POST /pet`

Create a new pet listing

Add a new pet to the store.

Request body

Field	Type	Required
`id`	integer (int64)	

`name`	string	yes
`category`	Category	
`photoUrls`	array of string	yes
`tags`	array of Tag	
`status`	string (available | pending | sold)	

Responses

Code	Description
`200`	Successful operation
`400`	Invalid input
`422`	Validation exception
`default`	Unexpected error

Actions 3 and 4 land on `GET /pet/findByStatus` — the **Deprecation** caution in its description and the enriched **status** parameter both come from the overlay, not the upstream spec:

Swagger Petstore - OpenAPI 3.0

v1.0.27

Browse and manage pets 🐾, store orders, and users — see the overlay guide. This reference is the upstream **Swagger Petstore** spec, customized entirely through an aardvark overlay.

Sandbox API

Every operation here runs against the public Petstore sandbox — data resets periodically, so experiment freely.

> **🔑Authentication**
>
> Most write operations need an API key — paste one into the **Authorize** box and it rides along with every Try it now call.

❖**Status codes**

Errors follow standard HTTP semantics: `4xx` for client problems, `5xx` for ours.

Base URL: `https://petstore3.swagger.io/api/v3`

`GET /pet/findByStatus`

Finds Pets by status.

Returns every pet matching the given **status** values — pass one or more of `available`, `pending`, or `sold` in the `status` query parameter.

Deprecation

The multi-value form is being phased out — prefer a single status per request.

Parameters

Name	In	Type	Required	Description
`status`	query	string (available | pending | sold)	yes	Filter by status — one or more of `available`, `pending`, or `sold`.

Responses

Code	Description
`200`	successful operation
`400`	Invalid status value
`default`	Unexpected error

Because overlays are applied to the whole document **before** any slice is taken, an overlaid summary, description, or parameter shows on a single-operation slice exactly as it does in a full-spec reference.

Options

The directive takes a quoted spec path and an optional `verb` / `endpoint` pair:

Attribute	Effect

`'…'` (first argument)	Path to the OpenAPI spec, relative to the **site root** (**required**). JSON or YAML.
`verb='get'`	HTTP method of the operation to slice (`get`, `post`, `put`, `patch`, `delete`, …). Use **with** `endpoint`; omit **both** to render the whole spec.
`endpoint='/pet/{petId}'`	Path of the operation to slice — exactly as it appears in the spec, leading slash and `{param}` placeholders included. Use **with** `verb`.

A page's `sortableTables` and `filterableTables` settings (or the site `tables` block) also govern the reference's tables.

How it works

The directive loads and parses the spec and emits the built-in `ApiReference` React component (an island) with the spec as a prop; slicing simply narrows the spec to one operation first. The "Try it now" form builds the request from your path and query parameters and runs `fetch` against the spec's `servers[0].url` (subject to the API's CORS policy). Security schemes ride along on that same prop, so the Authorization key and the per-language samples are computed in the browser — your key never leaves it.

With build-time AI enabled, Aardvark augments authored examples: when an operation's success response has no `example`, it generates a realistic one. Examples you write in the spec always win.

CSS Selectors

The reference mounts inside an island wrapper carrying `data-aardvark-island="ApiReference"` and renders its own class names — target the wrapper, each operation, its method badge and endpoint, the **Try it now** accordion, and the samples / responses rail.

```
[data-aardvark-island="ApiReference"]  /* the island wrapper */
.aardvark-api-op                       /* one operation section */
.aardvark-api-method                   /* the HTTP method badge */
.aardvark-api-endpoint                 /* the endpoint bar */
.aardvark-api-endpoint-url             /* the request path inside it (monospace) */
.aardvark-api-tryit                    /* the "Try it now" accordion */
.aardvark-api-rail                     /* the samples + responses rail */
```

Injecting Attributes

`{% openapi %}` is a self-contained directive, not a wrapped component, so it doesn't take a raw `attr={…}` channel — style it through the CSS parts above, and configure it with the documented attributes (`verb`, `endpoint`). To customize a spec you can't edit — rewrite a summary, inject a primitive, hide an operation — use an overlay instead.

Panel

`panel` is a **supplementary side panel** — a bordered, subtly-raised surface for secondary content that sits beside the main text. Use it for a "see also", an API note, related links, or any aside that supports the main flow without interrupting it. On a wide screen the panel floats to one side and the surrounding text wraps alongside it; on a narrow screen it drops into place as a full-width block. It ships with Aardvark, so a side panel is a single tag with no setup.

The block body is the panel content (any Markdown), and an optional `title` renders a small heading above it. Use it as `{% panel %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'panel', …)`. Close it with `{% endPanel %}`.

Demonstrations

A titled panel floats to the right by default, and the main content wraps around it:

See also

Related reading:

- The layout overview
- Paper surfaces
- The AppShell

The panel keeps secondary material close to the text it supports without pushing it out of the way. On a wide viewport it floats beside this paragraph; shrink the window and it stacks into a full-width block above this text instead. This is the whole point of a side panel — it stays out of the main reading path while remaining a glance away, and the responsive stacking means it never crushes the prose on a phone.

Source: Markdown

```
{% panel title="See also" %}
Related reading:

- [The layout overview](/components/layout/)
- [Paper surfaces](/components/layout/paper/)
- [The AppShell](/components/layout/appshell/)
{% endPanel %}
```

Source: Python

```
component(
    'aardvark', 'panel',
    title='See also',
    children=(
        'Related reading:\n\n'
        '- [The layout overview](/components/layout/)\n'
```

```
        '- [Paper surfaces](/components/layout/paper/)\n'
        '- [The AppShell](/components/layout/appshell/)'),
)
```

Float to the left

Set `side="left"` to float the panel to the left edge instead:

Note

A panel can float to **either** side — pick whichever keeps it nearest the content it supports.

The surrounding paragraph flows around whichever side the panel takes. Left-floating panels read well next to a list or a definition that the panel annotates, since the eye reaches the panel before the text it comments on. As with the default right float, the panel drops below the text as a full-width block once the viewport is too narrow to sit prose beside it.

Source: Markdown

```
{% panel title="Note" side="left" %}
A panel can float to **either** side — pick whichever keeps it nearest the content it
supports.
{% endPanel %}
```

Source: Python

```
component(
    'aardvark', 'panel',
    title='Note', side='left',
    children='A panel can float to **either** side — pick whichever keeps it nearest the
content it supports.',
)
```

Without a title

`title` is optional — omit it for a plain surface:

No heading here — just a raised surface for a quick aside.

Source: Markdown

```
{% panel %}
No heading here — just a raised surface for a quick aside.
{% endPanel %}
```

Source: Python

```
component('aardvark', 'panel',
          children='No heading here — just a raised surface for a quick aside.')
```

Attributes

Attribute	Valid values	Description
`title`	Any string	Optional heading shown above the body. Omit it for a plain surface.
`side`	`right` (default), `left`	Which edge the panel floats to on a wide viewport.
`width`	Any CSS length (e.g. `18rem`, `40%`)	Cap the floated panel's width. Defaults to a comfortable column.
`shadow`	`xs` (default) / `sm` / `md` / `lg` / `xl`	Drop shadow / elevation.
`radius`	`xs`–`xl`, or any CSS value	Corner rounding (default `md`).
`withBorder`	bare flag or `true` / `false` (default `true`)	The 1px border.
`p` / `px` / `py` / `pt` / `pb` / `pl` / `pr`	Spacing token or any CSS value	Padding (defaults to `lg`).
`attr`	`{…}`	Raw HTML attributes forwarded onto the rendered element (see below).
(body)	Markdown	The panel content, written between `{% panel %}` and `{% endPanel %}` (`children=` from Python).

CSS Selectors

Each `panel` carries `data-aardvark-island="Panel"` on its wrapper, and the rendered Mantine Paper exposes its root as `mantine-Paper-root` — target either to style it from your theme CSS.

```
[data-aardvark-island="Panel"] {
  /* style every panel on the page */
}

[data-aardvark-island="Panel"] .aardvark-panel-title {
  /* the optional heading */
}
```

```
[data-aardvark-island="Panel"] .aardvark-panel-body {
  /* the Markdown body */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Interactive

Click this panel to fire the handler.

Source: Markdown

```
{% panel title="Interactive" attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Click this panel to fire the handler.
{% endPanel %}
```

Source: Python

```
component(
    'aardvark', 'panel',
    title='Interactive',
    children='Click this panel to fire the handler.',
    attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
)
```

Prompt

A **built-in** tag that renders a copyable **AI-prompt block**: your prompt text in a tidy code surface with a **copy** button and one-click **“Open in …”** deep links that launch ChatGPT, Claude, or Cursor with the prompt already filled in. It’s ideal for docs that hand the reader a ready-to-run prompt — a “fix this with AI”, a scaffolding request, a review checklist — so they can send it to their assistant without copy-pasting by hand.

The body is the **literal prompt text**: it’s kept **verbatim** — never run through Markdown — so line breaks, blank lines, and any code or angle brackets survive exactly as written (that’s the text that gets copied and encoded into each deep link). The `title` is an optional heading, and `actions` chooses which buttons appear. Close it with `{% endPrompt %}`.

Use it as `{% prompt %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'prompt', …)`.

Usage

Put the prompt between `{% prompt %}` and `{% endPrompt %}`:

Review this pull request

You are a senior engineer reviewing a pull request. For each file in the diff:

1. Summarize what changed in one sentence.
2. Flag any correctness bugs, with the line and a suggested fix.
3. Note missing tests or docs.

Be concise and specific. Do not restate the diff.

Source: Markdown

```
{% prompt title="Review this pull request" %}
You are a senior engineer reviewing a pull request. For each file in the diff:

1. Summarize what changed in one sentence.
2. Flag any correctness bugs, with the line and a suggested fix.
3. Note missing tests or docs.

Be concise and specific. Do not restate the diff.
{% endPrompt %}
```

Source: Python

```
component('aardvark', 'prompt', title='Review this pull request',
          children=(
              'You are a senior engineer reviewing a pull request. '
              'For each file in the diff:\n\n'
              '1. Summarize what changed in one sentence.\n'
```

```
                '2. Flag any correctness bugs, with the line and a suggested fix.\n'
                '3. Note missing tests or docs.\n\n'
                'Be concise and specific. Do not restate the diff.'))
```

The **Copy** button copies the whole prompt to the clipboard; each **Open in …** button URL-encodes the prompt into that tool's "new chat" link — `https://chatgpt.com/?q=…`, `https://claude.ai/new?q=…`, and Cursor's `cursor://…?text=…` app deep link.

Choosing the buttons

`actions` is a comma-separated list of the buttons to show, in order. Valid values are `copy`, `chatgpt`, `claude`, and `cursor`; the default is all four (`copy, chatgpt, claude, cursor`). Pass a shorter list to trim or reorder them — here, just **Copy** and **Claude**:

Scaffold a REST endpoint

Add a new REST endpoint POST /widgets to this codebase.

- Validate the request body against the existing schema conventions.
- Persist with the current ORM/data layer.
- Return 201 with the created resource, 400 on validation errors.
- Include a unit test that covers the happy path and one validation failure.

Source: Markdown

```
{% prompt title="Scaffold a REST endpoint" actions="copy, claude" %}
Add a new REST endpoint POST /widgets to this codebase.

- Validate the request body against the existing schema conventions.
- Persist with the current ORM/data layer.
- Return 201 with the created resource, 400 on validation errors.
- Include a unit test that covers the happy path and one validation failure.
{% endPrompt %}
```

Source: Python

```
component('aardvark', 'prompt', title='Scaffold a REST endpoint',
          actions=['copy', 'claude'],
          children=(
              'Add a new REST endpoint POST /widgets to this codebase.\n\n'
              '- Validate the request body against the existing schema conventions.\n'
              '- Persist with the current ORM/data layer.\n'
              '- Return 201 with the created resource, 400 on validation errors.\n'
              '- Include a unit test that covers the happy path and one validation
failure.'))
```

Without a title

`title` is optional — omit it and the block shows a plain **Prompt** label above the text:

Summarize the following text in three bullet points, then give it a one-line title.

Source: Markdown

```
{% prompt %}
Summarize the following text in three bullet points, then give it a one-line title.
{% endPrompt %}
```

Source: Python

```
component('aardvark', 'prompt',
          children='Summarize the following text in three bullet points, then give it a
one-line title.')
```

Verbatim body

The body is **not** Markdown — it's copied exactly as written, so `**stars**`, `# hashes`, `<angle brackets>`, and fenced code all stay literal. If your prompt contains Aardvark template syntax (a literal `{% … %}` or `{{ … }}`), wrap that part in a **raw block** so the template engine leaves it alone — the same `{% raw %}` … raw-guard convention used everywhere else in these docs.

Options

Attribute	Valid values	Description
`title`	Any string	Optional heading shown above the prompt. Omit it for a plain **Prompt** label.
`actions`	Comma-separated list of `copy`, `chatgpt`, `claude`, `cursor`	Which buttons to show, in order. Default `copy, chatgpt, claude, cursor` (all four). Unknown values are ignored; an empty/all-unknown list falls back to just **Copy**.
`attr`	`{…}`	Raw HTML attributes forwarded onto the rendered block root (see below).
(body)	Verbatim text	The prompt itself, written between `{% prompt %}` and `{% endPrompt %}` (`children=` from Python). Kept exactly as written — never Markdown-rendered.

CSS Selectors

Each prompt carries `data-aardvark-island="Prompt"` on its wrapper and renders its own class names — target the block, the header, the buttons, or the prompt text.

```
[data-aardvark-island="Prompt"]               /* the island wrapper */
[data-aardvark-island="Prompt"] .aardvark-prompt        /* the bordered block */
[data-aardvark-island="Prompt"] .aardvark-prompt-head   /* the title / buttons row */
[data-aardvark-island="Prompt"] .aardvark-prompt-action /* a copy / open-in button */
[data-aardvark-island="Prompt"] .aardvark-prompt-text   /* the verbatim prompt <pre> */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered block root.

Explain this error

Explain the following error message in plain language, then list the two most likely causes and how to check each.

Source: Markdown

```
{% prompt title="Explain this error" attr={'data-analytics': 'ai-prompt', 'aria-label': 'AI
prompt: explain error'} %}
Explain the following error message in plain language, then list the two most likely causes
and how to check each.
{% endPrompt %}
```

Source: Python

```
component('aardvark', 'prompt', title='Explain this error',
          children='Explain the following error message in plain language, then list the two
most likely causes and how to check each.',
          attr={'data-analytics': 'ai-prompt', 'aria-label': 'AI prompt: explain error'})
```

Target

A **built-in** tag that names a region of a page:

```
{% target id="install-cli" %}
Everything in here is one section.
{% endTarget %}
```

Content Reach already treats every `##` and `###` heading as a section, which covers your prose. `{% target %}` covers the blocks that matter most but aren't headings — the install command, the API auth step, the "Try it now", the sign-up call to action.

The tag renders nothing of its own: no box, no spacing, no styling. The region looks exactly as it did before you wrapped it.

The id

Required, and a lowercase slug: letters, digits, `-` and `_`, starting with a letter or digit, up to 64 characters. That is the same shape as a generated heading anchor, on purpose — the dashboard shows one ordered list of sections rather than two parallel ones.

A malformed or missing id emits a build warning and renders the body **unwrapped**. You lose the measurement for that region until the id is fixed; you never lose the content.

Ids should be stable. Renaming one starts a new section in the dashboard rather than continuing the old one's history, the same way renaming a heading does.

A worked example

````
## Getting started

{% target id="install-cli" %}
Install the CLI:

```bash
npm install -g aardvark
```
{% endTarget %}

Then run `vark new` to scaffold a site.
````

The page now reports three sections: the `getting-started` heading, the `install-cli` target inside it, and whatever heading follows. Copying that `npm install` line is attributed to `install-cli` rather than to the whole `getting-started` section.

Markdown inside

The body is ordinary Markdown — headings, lists, code fences, other components — and renders exactly as it would outside the tag. Nesting a heading inside a target is fine; both become sections, in document order — which is also how their time divides. The target is credited from its opening tag down to that heading, and the heading from there on, since it is the narrower region the reader is in. Both still count as reached when they come into view. If you want the whole region's time under one id, put the target inside the heading's section rather than around it.

A single component

For one component rather than a region, no new tag is needed. The generic `attr=` channel every built-in accepts carries the same marker:

```
{% button url="/signup" attr={"data-aardvark-target": "signup-cta"} %}Start free{% endButton %}
```

Reach for `{% target %}` when you want a region; reach for `attr=` when you want one element.

What it does not do

- **No effect without Content Reach.** With the feature off, or on a plan that doesn't include it, the marker is inert markup — harmless, and costing nothing.
- **Nothing reaches the agent Markdown.** The wrapper is stripped from each page's `.md` twin and from `llms-full.txt`, so an AI agent reading your docs never sees it.
- **It captures no content.** Marking a region records that the region was on screen and that its controls were used, never anything inside it.

Taxonomy

A **built-in** tag that renders a listing of **member pages** — real pages in your content tree that opt into a named taxonomy through front matter — in one of three shapes: a **changelog** timeline, a **knowledge base** of categorized questions, or a **blog**-style article list. Because the members are ordinary pages, every entry has its own URL and its own Markdown body — and, unless it opts out with `noindex: true`, its own place in search and `llms.txt`. This site's **changelog** entries do opt out: they're stealthed (`nav: false` + `noindex: true`), and the changelog listing folds their body text into its own search record, so it stays findable there. That fold is specific to the changelog type — **blog** and **knowledge-base** members are ordinarily their own search landings, so a listing there carries only their teaser; marking such a member `noindex: true` drops its body from search rather than recovering it through the listing. The listing is assembled at build time from their front matter. This site uses all three: the Changelog tab, the Support knowledge base, and the Blog are each one `{% taxonomy %}` tag over a folder of member articles. Changelog and article listings also publish an **RSS feed** (pass `feed=false` to opt out); knowledge-base listings do not.

Usage

Two halves: member pages declare their membership in front matter, and one listing page renders the tag. A member joins a taxonomy with a `taxonomy:` entry:

```
---
title: Faster dev-loop rebuilds
date: 2026-05-28 09:15
taxonomy:
  - name: changes
    tags: [build, performance]
---
```

and the listing page — the one page marked `listing: true` — renders all members:

```
{% taxonomy name="changes" type="changelog" wide %}
```

The member's **body** is the entry's content — for `type="changelog"` it renders inline in the timeline exactly as `{% changelog %}` renders a YAML entry's description, so a data-file changelog can convert to member articles with an identical rendered result. (Island components in a member body are suppressed in the inline listing view — they render only on the member's own page.) Relative links in a member body are rebased onto the member's own URL, so they resolve the same inline as on the article page; on a versioned site, a root-absolute link in an older version's listing is rewritten to that version's equivalent page when one exists — the same in-version rule the member's own page follows — so the inline view never bounces a reader to the latest docs.

Front matter

Member pages

Each entry in a page's `taxonomy:` list joins one taxonomy. `name` is required; everything else is optional:

Key	Effect
`name`	The taxonomy this page joins (**required**). `taxonomyName` is accepted as an alias — prefer `name`.
`tags`	Labels for this entry — a list (`[a, b]`) or a comma-separated string (`"a, b"`). These drive the filter cloud, KB categories, and `tagFilter`, and are listed at the bottom of the member's own page as links into the listing's filtered view.
`listing`	`true` marks this page as the taxonomy's canonical listing page (the one carrying the `{% taxonomy %}` tag).
`leftnav`	`true` shows a taxonomy tag navigation in this page's left sidebar (a non-clickable section heading over the tag links). `dates` instead shows a **month-grouped member archive** — the articles themselves, newest first, under "June 2026"-style headings — replacing the menu's alphabetical entries (this site's Blog does this).
`articleCount`	`true` adds per-tag article counts to those labels (as small badge pills).
`tagCloud`	`true` shows the taxonomy's tag cloud in this page's right rail. Not available on wide/full-mode pages (they have no right rail).
`featured`	`true` flags a knowledge-base entry as a **top question**, surfaced above the categories. `topQuestion` is accepted as an alias.
`authorName`	Blog byline shown on the article card **and at the top of the article page itself** (avatar + name + publish date, tucked under the title — the page's bottom date line then keeps only "Last modified"). A **dated** member without an author still gets the date-only byline — this site's changelog entries show their publish date under the title this way.
`authorAvatar`	Avatar image for the byline. Omit it and both bylines fall back to the author's initials.
`badgeText`	A badge label on the article card.

Several **top-level** front-matter keys also feed the entry: `title` (the entry headline), `date` (`YYYY-MM-DD`, optionally with a time — `2026-05-28 16:30` — the full timestamp drives the newest-first sort), `description` (the card / KB excerpt text), `version` (a changelog release badge beside the date), and `image` / `imageAlt` (a blog card cover image and its alt text — omit `imageAlt` when the cover is decorative).

The listing page

Give exactly one page `listing: true` for the taxonomy and put the tag in its body. A changelog- or blog-style listing usually wants a wide layout (`mode: full` or `mode: wide`) so it can use `wide`.

Options

Attribute	Effect
`name="…"`	The taxonomy to render (**required**) — matches the members' `name`.
`type="…"`	The listing shape: `changelog` (timeline), `kb` (knowledge base), or `articles` (blog-style cards; `blog` is an alias).
`tagFilter="…"`	Show only members carrying this tag (case-insensitive). A comma-separated value matches members carrying **any** listed tag — unless the whole value exactly matches a single tag whose own name contains a comma (`tagFilter="Plans, seats & hosting"`), which wins.
`cardVariant="…"`	Card look for article listings: `horizontal`, `footer` (alias `withfooter`), `background` (alias `image`), `vertical`, or `plain` (alias `grid`).
`wide`	Lay the tag cloud out in a sticky right-hand column. **Requires a wide layout mode** — `mode: wide`, `full`, or `uncapped` in the page front matter (see Modes).
`limitEntries=20`	Show at most N entries (after the newest-first sort). `limit` is an alias.
`limitDays=90`	Changelog type only — show only entries from the last N days, counted from the build date.
`combineByDay`	Changelog type only — merge all of a day's entries into one timeline entry. The day becomes the atomic item, so the tag cloud / `#tag=` filter then works at day granularity (selecting a tag keeps any day that carries it, showing that day's other entries too).
`cols=3`	Column count for article-card grids, on desktop. Like `{% cardGrid %}`, the grid steps down on its own — two columns on a tablet, one on a phone — so the cards wrap instead of shrinking.
`topCount=5`	Knowledge base only — how many featured **top questions** to surface (`topCount=0` drops the section).
`emptyText="…"`	The no-matches message, on every listing type. Override it to localize the listing's chrome.

`filterLabel=` / `clearLabel=`	Changelog/articles — the tag cloud's "Filter by tag" heading and its "Clear" button. Override to localize.
`rssLabel="…"`	Changelog/articles — the RSS link's accessible name and tooltip (default "RSS feed"). Override to localize.
`topLabel=` / `categoriesLabel=` / `allLabel=` / `showAllLabel=`	Knowledge base only — the three section headings ("Top questions", "Browse by category", "All articles") and the filtered view's "Show all" clear link (while a filter is active, the articles heading names the active tag instead of `allLabel`). Override to localize.
`articleLabel=` / `articlesLabel=`	Knowledge base only — the singular/plural noun in the "N article(s)" count line (category-card counts and the filtered status line). Override to localize the count alongside the headings above.
`banner=false`	Knowledge base only — drop the **hero banner**. By default a KB listing opens with a banner holding an "ask your question" field: the question goes to the site's AI assistant when one is enabled, else it opens **search** pre-filled. With neither surface enabled the banner drops on its own.
`bannerTitle=` / `bannerText=` / `askPlaceholder=` / `askLabel=`	Knowledge base only — the banner's heading (default "How can we help?"), optional subtitle, the ask field's placeholder, and its submit button label. Override to localize.
`toc=false`	Changelog type only — don't add per-entry headings to the page's "On this page" list (added by default in a non-wide layout; article and KB listings never contribute TOC entries).
`feed=false`	Changelog and article listings only — don't publish an RSS feed (one is emitted for each by default, advertised in the page's `<head>`). KB listings never publish a feed.
`attr={…}`	Forward raw HTML attributes onto the rendered root — see below.

A changelog listing

`type="changelog"` reproduces the `{% changelog %}` timeline — the same component, fed from member articles instead of a YAML file. This is exactly what the Changelog tab renders (with `wide`); here, capped to the three most recent entries:

```
{% taxonomy name="changes" type="changelog" limitEntries=3 toc=false feed=false %}
```

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Standardized code-block Copy & Download buttons — May 22, 2026

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

A knowledge base

`type="kb"` groups members by tag into categories, with `featured: true` members surfaced as top questions and each entry's `description` as its excerpt. The Support tab is one of these; here, filtered to a single category:

```
{% taxonomy name="support" type="kb" tagFilter="Included AI" %}
```

Top questions

What happens when I use up my included AI for the month?

By default paid AI pauses and readers fall back to free models, so you're never charged a surprise overage.

What if I don't use all of it — does it roll over?

Unused allowance rolls over up to one month's worth, so allowance plus rollover never exceeds about two months at once.

All articles

What exactly counts against the included AI?

The billed dollar cost of metered requests counts against your allowance, pooled account-wide and drained before your prepaid balance.

What gets used first — my allowance or my prepaid balance?

Rolled-over allowance drains first, then this month's allowance, and your prepaid balance only if you've opted into overflow.

What happens when I use up my included AI for the month?

By default paid AI pauses and readers fall back to free models, so you're never charged a surprise overage.

What if I don't use all of it — does it roll over?

Unused allowance rolls over up to one month's worth, so allowance plus rollover never exceeds about two months at once.

An article list

`type="articles"` renders blog-style article cards — byline, date, badge, cover image, and the `description` as the teaser. The Blog tab renders all posts (with `wide`); here, filtered to one tag:

```
{% taxonomy name="blog" type="articles" tagFilter="release" feed=false %}
```

vark 0.9 release roundup

The 0.9 release in one read — a self-generating CLI reference, a faster dev loop, and the best of the 0.8 line.

Traversing a taxonomy yourself

The three listing types are bespoke renderings, but the underlying data is plain and yours to walk: every `{% %}` block runs real Python, and two injected helpers open the taxonomy up — **`taxonomy(name)`** returns the named taxonomy for the current page's language/version, and **`member_html(member)`** renders a member's actual body content (see below). Loop with `print()` to emit any markdown (or HTML) you like:

```
{%
for m in taxonomy("changes").articles:
    print(f"- [{m.title}]({m.url}) — {m.date_display}\n")
%}
```

which renders the changelog members as a plain list:

- Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM
- Faster dev-loop rebuilds — May 28, 2026, 9:15 AM
- Standardized code-block Copy & Download buttons — May 22, 2026
- Index OpenAPI descriptions for search — May 14, 2026
- Ask-AI reader assistant — May 3, 2026
- Build-time accessibility contrast audit — April 19, 2026
- Social unfurl cards (Open Graph) — April 5, 2026
- Multi-language sites — March 21, 2026
- Interactive Mantine islands — March 2, 2026

The returned object exposes the taxonomy's whole surface:

Accessor	What you get
`.articles`	The member pages minus the listing page, sorted newest-first (undated members last) — each with `.title`, `.url`, `.description`, `.tags`, `.date_display`, `.when` (a `datetime` or `None`), `.version`, `.featured`, `.author_name`, `.author_avatar`, `.badge_text`, `.image`.
`.members`	The same list including the listing page (`.is_listing` tells them apart).
`.by_tag`	`{tag: [members]}` — group by tag, count per tag, build your own category index.
`.listing_url`	The canonical listing page's URL.

An unknown name raises at build time, listing the taxonomies that exist — the same contract as the directive. Combine with `featured`/`by_tag` for custom surfaces the built-ins don't cover ("three most recent posts per tag", a by-author archive, …). Here's this site's real Support taxonomy, grouped by tag:

```
{%
for tag, members in sorted(taxonomy("support").by_tag.items()):
    print(f"**{tag}** — {len(members)} article(s)\n")
%}
```

Included AI — 4 article(s) **Overflow** — 3 article(s) **Payments & limits** — 8 article(s) **Plan changes** — 4 article(s) **Plans, seats & hosting** — 8 article(s)

Member content

Front matter is only the surface — `member_html(member)` renders a member's actual **body** to HTML, so a loop can reproduce the articles themselves, not just link to them. The body renders in the member's own context (includes and expressions resolve exactly as on the article page), the leading H1 is stripped so your loop prints its own headline, and relative links are rebased onto the member's URL. Island components are suppressed with a build warning — they render only on the member's own page, the same contract as the inline changelog view. Calling it from inside a member's own body is rejected (a member embedding other members could recurse). Here are this site's two most recent release notes, re-rendered in full:

```
{%
for m in taxonomy("changes").articles[:2]:
    print(f"\n#### {m.title} — {m.date_display}\n")
    print(member_html(m))
%}
```

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

CSS Selectors

Each listing shape renders its own class-name family — target the article list, the knowledge base, the tag cloud, and the changelog timeline through their prefixes:

```
.aardvark-articlelist-*   /* article-list layout and cards */
.aardvark-kb-*            /* knowledge-base categories, top questions, entries */
.aardvark-tagcloud-*      /* the tag cloud (listing column or right rail) */
.aardvark-changelog-*     /* the changelog timeline (same parts as {% changelog %}) */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered listing root:

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Source: Markdown

```
{% taxonomy name="changes" type="changelog" limitEntries=2 toc=false feed=false
attr={'data-analytics': 'release-feed', 'aria-label': 'Changelog'} %}
```

Source: Python

```
component('aardvark', 'taxonomy', name='changes', type='changelog', limitEntries=2,
          toc=False, feed=False,
          attr={'data-analytics': 'release-feed', 'aria-label': 'Changelog'})
```

Update

The built-in **update** tag renders a single **release-note** — one entry on a timeline rail: a version or date **label** on the left, and the entry's **Markdown** body beside it. Give it a `label` (the version or date), an optional `description` (a dimmed sub-label — a date under a version, say), and an optional `tags` cloud. Close it with `{% endUpdate %}`.

Reach for it when you want to hand-author a short **"What's new"** note on a page, or stack a run of them into a release log. It is **distinct** from the Changelog tag: `{% changelog %}` reads a whole **YAML data file** and owns the tag-filter cloud and RSS feed, while `{% update %}` is a **single inline entry** you write in the page.

Use it as `{% update %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'update', …)`.

Demonstrations

A single entry

A `label` on the rail, the Markdown body beside it:

v1.4.0

Added a **dark-mode** toggle and a per-page table of contents. See the theming guide for the new CSS variables.

Source: Markdown

```
{% update label="v1.4.0" %}
Added a **dark-mode** toggle and a per-page table of contents. See the
[theming guide](/theming/) for the new CSS variables.
{% endUpdate %}
```

Source: Python

```
component('aardvark', 'update', label='v1.4.0', children=(
    'Added a **dark-mode** toggle and a per-page table of contents. See the '
    '[theming guide](/theming/) for the new CSS variables.'))
```

A run of entries (a release log)

Stack several — each is its own entry, and the rail joins them into one continuous timeline. Entries render in the order you write them, so put the newest first:

v2.1.0

- New `search` endpoint with typo-tolerant matching.
- The dashboard now remembers your last filter.

v2.0.1

Fixed a race in the billing reconciler that could double-count a refund.

v2.0.0

Renamed the `token` parameter to `apiKey` across every endpoint. See the migration notes before upgrading.

Source: Markdown

```
{% update label="v2.1.0" description="2026-07-03" tags="feature, api" %}
- New `search` endpoint with typo-tolerant matching.
- The dashboard now remembers your last filter.
{% endUpdate %}

{% update label="v2.0.1" description="2026-06-18" tags="fix" %}
Fixed a race in the billing reconciler that could double-count a refund.
{% endUpdate %}

{% update label="v2.0.0" description="2026-06-01" tags="breaking" %}
Renamed the `token` parameter to `apiKey` across every endpoint. See the
[migration notes](/) before upgrading.
{% endUpdate %}
```

Source: Python

```
component('aardvark', 'update', label='v2.1.0', description='2026-07-03',
          tags='feature, api', children=(
              '- New `search` endpoint with typo-tolerant matching.\n'
              '- The dashboard now remembers your last filter.'))
component('aardvark', 'update', label='v2.0.1', description='2026-06-18',
          tags='fix', children=(
              'Fixed a race in the billing reconciler that could double-count a refund.'))
component('aardvark', 'update', label='v2.0.0', description='2026-06-01',
          tags='breaking', children=(
              'Renamed the `token` parameter to `apiKey` across every endpoint. See the '
              '[migration notes](/) before upgrading.'))
```

A custom bullet and accent color

`bullet` sets the rail glyph — a Tabler icon name (lazily fetched), an emoji, or short text — and `color` accents this entry's bullet and line:

v1.5.0

Shipped the new onboarding flow.

Source: Markdown

```
{% update label="v1.5.0" bullet="rocket" color="grape" %}
Shipped the new onboarding flow. 🚀
{% endUpdate %}
```

Source: Python

```
component('aardvark', 'update', label='v1.5.0', bullet='rocket', color='grape',
          children='Shipped the new onboarding flow. 🚀')
```

Attributes

Attribute	Valid values	Description
`label`	Any string	The rail heading — the version or date (e.g. `v1.2.0`, `2026-07-03`). Optional; omit for an unlabelled entry.
`description`	Any string	An optional dimmed sub-label under the label (a date beneath a version, say).
`tags`	Comma-separated string	An optional badge cloud under the label (e.g. `tags="feature, fix"`).
`bullet`	Tabler icon name, emoji, or short text	The rail glyph. A bare Tabler name is fetched lazily; anything else renders verbatim; unset uses Mantine's default dot.
`color`	Any Mantine color	Accent for this entry's bullet and line. Defaults to your site's primary color.
`attr`	`{…}`	Raw HTML attributes forwarded onto the rendered island root (see below).
(body)	Markdown	The entry content, written between `{% update %}` and `{% endUpdate %}` (`children=` from Python).

Update vs. Changelog

Both draw a timeline, but they solve different problems:

	`{% update %}`	**`{% changelog %}`**
Source of entries	Authored inline, one tag per entry	A YAML data file

Best for	A short “What’s new” note, a hand-written release log	A full, filterable changelog with a feed
Tag filtering	A static badge cloud (display only)	An interactive filter cloud
RSS feed / On-this-page	No	Yes

CSS Selectors

Each `update` carries `data-aardvark-island="Update"` on its wrapper, and the rendered Mantine Timeline exposes its parts as `mantine-Timeline-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Update"] {
  /* style every update entry on the page */
}

.aardvark-update-version {
  /* the version/date label on the rail */
}

.aardvark-update-body {
  /* the Markdown body */
}

.mantine-Timeline-itemBullet {
  /* the rail bullet */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered island root.

v1.6.0

Added keyboard shortcuts across the app.

Source: Markdown

```
{% update label="v1.6.0" attr={'data-analytics': 'release-note', 'aria-label': 'v1.6.0
release note'} %}
Added keyboard shortcuts across the app.
{% endUpdate %}
```

Source: Python

```
component('aardvark', 'update', label='v1.6.0',
          children='Added keyboard shortcuts across the app.',
          attr={'data-analytics': 'release-note', 'aria-label': 'v1.6.0 release note'})
```

Visibility

A **built-in** block tag that shows or hides a chunk of Markdown depending on who is reading the page: a **human** looking at the rendered HTML in a browser, or an **AI agent** fetching the page's plain-Markdown twin.

```
{% visibility for="agent" %}
This paragraph is written for an AI agent — it appears in the page's `.md`,
but is hidden from readers of the HTML page.
{% endVisibility %}
```

This is the same idea as Mintlify's Visibility / View control, and a natural fit for Aardvark's **Markdown for Agents** story: every page is served both as rendered HTML and — to a client that asks for `text/markdown` — as a prebuilt `.md` representation. `{% visibility %}` lets one source file carry a little extra guidance for agents, or hide browser-only chrome from them, without maintaining two copies of the page.

Modes

Set `for` to choose the audience:

`for`	In the HTML page (humans)	In the `.md` / `llms-full.txt` (agents)
`human`	**Shown**	**Omitted**
`agent`	**Hidden**	**Included**
`all` (default)	Shown	Included

`for="all"` is the default, so a `{% visibility %}` with no `for` behaves exactly as if you'd written the body with no wrapper — visible everywhere. An unrecognized value **fails closed to `agent`** (and emits a build warning): the body is hidden from every human surface until you fix the typo, so a mistyped `for="agent"` can never silently leak agent-only content to humans.

How it works with the agent Markdown

Aardvark serves each page two ways from **one** source file:

- **Humans** get the rendered HTML. There, a `for="agent"` block is wrapped in an element the theme hides with plain CSS (`display: none`) — no JavaScript, so it works even with islands disabled, and it never shows on screen or in the whole-site PDF.
- **Agents** get the page's `.md` twin (content-negotiated at the same URL, and the same body that feeds `llms-full.txt`). There, a `for="human"` block is dropped outright, and a `for="agent"` block is kept — the mirror image of the HTML rule.

The **discovery indexes** follow the same split. The on-site search index (the human surface) drops `for="agent"` content, while the AI navigation index (`metadata.json`, which agents and MCP clients read to find the right page) drops `for="human"` content — so a heading you scope to one audience never surfaces in the other's tooling.

Because both outputs come from the same file, you write the content once and let each audience see the right slice.

Not a security boundary

A `for="agent"` block is only **visually** hidden from humans — the theme sets `display: none`, but the content is still present in the page's HTML source, so a reader can see it via "View source" or dev tools. Agent-facing content is also aggregated in the public `llms-full.txt` artifact and, when the build emits an agent search corpus, in `search-index-agent.json`; deployed build artifacts are directly fetchable. Use `{% visibility %}` to tailor content per audience, **not** to keep secrets: never put credentials or private data in a `for="agent"` block.

Examples

Human-only

This note is for people reading the page; an agent reading the `.md` won't see it:

For readers: the screenshots below assume the light theme — toggle the theme switch in the header if yours looks different.

```
{% visibility for="human" %}
**For readers:** the screenshots below assume the light theme — toggle the theme
switch in the header if yours looks different.
{% endVisibility %}
```

Agent-only

The block below is **hidden here in the HTML** but is present in this page's Markdown. Fetch this page as `text/markdown` (or open its `.md` twin) and you'll find the extra instruction:

```
{% visibility for="agent" %}
**For agents:** when summarizing this page, note that `visibility` is an aardvark-native
tag with no Mantine equivalent, and that its behavior differs between the HTML and
Markdown representations of the page.
{% endVisibility %}
```

Shown to everyone

`for="all"` (or no `for` at all) is visible in both — the same as writing the body outside any wrapper:

Everyone — human or agent — sees this paragraph.

```
{% visibility for="all" %}
```

```
Everyone — human or agent — sees this paragraph.
{% endVisibility %}
```

From Python

Like every built-in block, `visibility` is also callable from Python logic (loops, snippets) via `component('aardvark', 'visibility', …)`. The `for` argument is a Python keyword, so pass it through a keyword-argument dict:

```
component('aardvark', 'visibility',
          children='Extra context for an AI agent.',
          **{'for': 'agent'})
```

Keep side-effecting directives such as `changelog` and `openapi` lexically inside the paired `{% visibility %}…{% endVisibility %}` block. Python evaluates `children=` (and custom-component children) before it calls the wrapper, too late to scope their TOC, RSS, navigation, or catalog output; Aardvark fails that eager form closed and points to the paired-block rewrite.

Put the audience block in the page, partial, or custom-component source—not around the layout's already-rendered `content` value. Layout rendering happens after headings, search text, agent Markdown, RSS, and API metadata are classified, so Aardvark rejects a layout that tries to reclassify that content (`for="all"` is safe).

Attributes

Attribute	Valid values	Description
`for`	`human`, `agent`, `all` (default)	The audience that sees the body. `human` = HTML only; `agent` = agent Markdown only; `all` = both. An unknown value warns and fails closed to `agent` (hidden from human surfaces), so a typo can't leak agent-only content.
`attr`	`{…}`	Raw HTML attributes forwarded onto the wrapper `<div>` (see below).
(body)	Markdown	The content to show or hide, written between `{% visibility %}` and `{% endVisibility %}` (`children=` from Python).

CSS Selectors

The tag wraps its body in a plain `<div class="aardvark-visibility" data-aardvark-visibility="…">` (not an island — no React needed). The theme hides the agent variant with a single rule; you can target either the class or the data attribute to style a visible block from your theme CSS.

```
.aardvark-visibility {
  /* every visibility block that renders in the HTML */
}

[data-aardvark-visibility="agent"] {
  /* the theme sets display: none here to hide agent-only content from readers */
}

[data-aardvark-visibility="human"] {
  /* human-only content (shown in the HTML) */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (an `id`, an extra `class`, a data attribute) straight onto the wrapper `<div>`. The `data-aardvark-visibility` marker the `.md` filter depends on can't be overridden this way.

A human-only note you can deep-link to with #reader-note.

```
{% visibility for="human" attr={'id': 'reader-note'} %}
A human-only note you can deep-link to with #reader-note.
{% endVisibility %}
```

Layout

Built-in tags for arranging content on the page — the Mantine layout primitives, each exposed as a single tag with no setup. Reach for these to put things in a row or a column, center them, cap a content width, or build a responsive grid.

Tag	What it does
Group	A flex **row** with consistent gaps.
Stack	A flex **column** with consistent gaps.
Flex	A flexbox container with the full CSS flex surface.
Center	Centers content horizontally and vertically.
Container	Caps content width with a comfortable gutter.
Grid	A flexible 12-column grid with spanning cells.
SimpleGrid	A responsive grid of equal-width columns.
AspectRatio	Keeps content at a fixed width-to-height ratio.
Space	An empty spacer between elements.
AppShell	A page-level shell (header / navbar / aside / footer).
Splitter	Two panels side by side with a divider.

Each tag forwards the matching Mantine props, so anything you can do with the underlying component is reachable from the tag. For the full prop surface of any one, follow its link to the Mantine docs at the top of each page.

Surfaces & utilities

The base primitive, card-like surfaces, scroll containers, and a few low-level helpers:

Tag	What it does
Box	The base primitive every component is built on — padding, margin, background, color, sizing, or a raw style on any block.

Paper	A card-like surface with a shadow, rounded corners, padding, and an optional border.
ScrollArea	A scrollable region with styled, auto-hiding overlay scrollbars.
Scroller	A minimal native-scroll container (the browser’s own scrollbars).
Scroll Buttons	A horizontal scroller with prev/next chevron buttons for paging a wide row of content.
Collapse	Animate content open and closed by height.
Marquee	A continuously scrolling ticker for logos, announcements, or a tagline.
Portal	Render content into a different part of the DOM to escape overflow and stacking contexts.

AppShell

`{% appshell %}` is a **built-in** tag for a **page-level layout shell**. It positions a header, navbar, aside, and footer around a main content area, the way an application's chrome wraps its content. Each region is opt-in — you add one by setting its text param — and the block body is the main content. Reach for it when you're scaffolding an app-style layout or a self-contained demo of one.

Use it as `{% appshell %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'appshell', …)`.

A self-contained frame

AppShell is a page-level primitive whose header, navbar, aside, and footer are normally fixed to the viewport. On an Aardvark page (which already has its own header and sidebar) that would take over the whole page, so the tag renders the shell inside a **bounded, self-contained frame** — each live example below sits in its own box and never escapes it. Size the frame with `height` (pixels, default 380).

Header and navbar

Set the `header` / `navbar` text params to add those regions; the block body is the main content area, which sits to the right of the navbar and below the header. Sizes are in pixels.

The main content area sits to the right of the navbar and below the header.

Source: Markdown

```
{% appshell header='My App' navbar='Navigation' headerHeight=56 navbarWidth=200 padding='md'
%}
The main content area sits to the right of the navbar and below the header.
{% endAppshell %}
```

Source: Python

```
component('aardvark', 'appshell', header='My App', navbar='Navigation',
          headerHeight=56, navbarWidth=200, padding='md',
          children='The main content area sits to the right of the navbar and below the
header.')
```

All four regions

Add `aside` (a right column) and `footer` (a bottom bar) the same way. A region only renders, and only reserves its size, once its text param is set.

Header, navbar, aside, and footer all framing the main content.

Source: Markdown

```
{% appshell header='Dashboard' navbar='Menu' aside='Details' footer='© 2026' headerHeight=48
footerHeight=40 navbarWidth=160 asideWidth=160 padding='md' %}
Header, navbar, aside, and footer all framing the main content.
{% endAppshell %}
```

Source: Python

```
component('aardvark', 'appshell', header='Dashboard', navbar='Menu',
          aside='Details', footer='© 2026', headerHeight=48, footerHeight=40,
          navbarWidth=160, asideWidth=160, padding='md',
          children='Header, navbar, aside, and footer all framing the main content.')
```

Layout mode

`layout` controls how the regions stack. `default` (the default) offsets the main content so the header, navbar, and aside frame it; `alt` makes the header and footer span the full width, with the navbar and aside tucked beneath the header.

With layout='alt', the header and footer run edge to edge.

Source: Markdown

```
{% appshell header='Full-width header' navbar='Nav' footer='Full-width footer' layout='alt'
headerHeight=48 footerHeight=40 navbarWidth=160 padding='md' %}
With layout='alt', the header and footer run edge to edge.
{% endAppshell %}
```

Source: Python

```
component('aardvark', 'appshell', header='Full-width header', navbar='Nav',
          footer='Full-width footer', layout='alt', headerHeight=48,
          footerHeight=40, navbarWidth=160, padding='md',
          children="With layout='alt', the header and footer run edge to edge.")
```

Borders off

Each region draws a border by default. Set `withBorder` to `false` to remove them for a flatter, seamless look.

No dividers between the regions.

Source: Markdown

```
{% appshell header='Borderless' navbar='Nav' headerHeight=48 navbarWidth=160 padding='md'
withBorder=false %}
No dividers between the regions.
```

```
{% endAppshell %}
```

Source: Python

```
component('aardvark', 'appshell', header='Borderless', navbar='Nav',
          headerHeight=48, navbarWidth=160, padding='md', withBorder=False,
          children='No dividers between the regions.')
```

With other components

The block body is ordinary Markdown, so you can drop other components into the main area — a `{% stack %}` of buttons, a badge, a callout, whatever the page needs.

[Profile]

Save changes

Source: Markdown

```
{% appshell header='Settings' navbar='Sections' headerHeight=48 navbarWidth=180 padding='md'
%}
{% stack gap='sm' %}
{% badge size='lg' %}Profile{% endBadge %}
{% button text='Save changes' %}
{% endStack %}
{% endAppshell %}
```

Source: Python

```
body = (
    component('aardvark', 'stack', gap='sm',
              children=component('aardvark', 'badge', size='lg', children='Profile')
                       + component('aardvark', 'button', text='Save changes'))
)
component('aardvark', 'appshell', header='Settings', navbar='Sections',
          headerHeight=48, navbarWidth=180, padding='md', children=body)
```

Attributes

A region is only rendered (and only reserves its size) when its text param is set. Omit anything else to take its Mantine default.

Attribute	Values	Description
`header`	any text	Header region text. Setting it adds a top bar.
`navbar`	any text	Navbar region text. Setting it adds a left column.

`aside`	any text	Aside region text. Setting it adds a right column.
`footer`	any text	Footer region text. Setting it adds a bottom bar.
`headerHeight`	a number, px (`60` default)	Height of the header region.
`footerHeight`	a number, px (`60` default)	Height of the footer region.
`navbarWidth`	a number, px (`250` default)	Width of the navbar column.
`asideWidth`	a number, px (`250` default)	Width of the aside column.
`padding`	`xs`–`xl` or any CSS value	Padding around the main content area.
`layout`	`default`, `alt`	`default` offsets Main inside the regions; `alt` makes header/footer span full width.
`withBorder`	`true`, `false` (default `true`)	Draw a border on each region. Set `false` to remove.
`height`	integer (pixels, default `380`)	Height of the self-contained frame the shell renders inside, so it stays a bounded preview instead of taking over the page.

CSS Selectors

Each `appshell` carries `data-aardvark-island="AppShell"` on its wrapper, and Mantine exposes its parts as `mantine-AppShell-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="AppShell"] {
  /* style every appshell on the page */
}

.mantine-AppShell-root {
  /* the root part */
}

.mantine-AppShell-header {
  /* the header part */
}
```

```
.mantine-AppShell-navbar {
  /* the navbar part */
}

.mantine-AppShell-main {
  /* the main part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

The main content area sits to the right of the navbar and below the header.

Source: Markdown

```
{% appshell header='My App' navbar='Navigation' headerHeight=56 navbarWidth=200 padding='md'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
The main content area sits to the right of the navbar and below the header.
{% endAppshell %}
```

Source: Python

```
component('aardvark', 'appshell', header='My App', navbar='Navigation',
          headerHeight=56, navbarWidth=200, padding='md',
          children='The main content area sits to the right of the navbar and below the
header.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

AspectRatio

`{% aspectratio %}` is a **built-in** tag that keeps its content at a **fixed width-to-height ratio** as it scales. It's the reliable way to embed a video, map, or image so the box never jumps the layout while the media loads. The `ratio` is a plain number — width divided by height — and the block body is the content that fills the box.

Use it as `{% aspectratio %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'aspectratio', …)`.

Widescreen (16:9)

`1.7777` is the classic 16:9 widescreen video ratio. Cap the size with `maw` (max width) so the box doesn't stretch edge to edge.

16 : 9

Source: Markdown

```
{% aspectratio ratio=1.7777 maw=480 %}
{% center h='100%' bg='var(--mantine-color-gray-2)' %}16 : 9{% endCenter %}
{% endAspectratio %}
```

Source: Python

```
inner = component('aardvark', 'center', h='100%',
                  bg='var(--mantine-color-gray-2)', children='16 : 9')
component('aardvark', 'aspectratio', ratio=1.7777, maw=480, children=inner)
```

A square (1:1)

`ratio=1` makes a perfect square — useful for avatars, thumbnails, and icon tiles.

1 : 1

Source: Markdown

```
{% aspectratio ratio=1 maw=200 %}
{% center h='100%' bg='var(--mantine-color-gray-2)' %}1 : 1{% endCenter %}
{% endAspectratio %}
```

Source: Python

```
inner = component('aardvark', 'center', h='100%',
                  bg='var(--mantine-color-gray-2)', children='1 : 1')
component('aardvark', 'aspectratio', ratio=1, maw=200, children=inner)
```

Standard (4:3)

`1.3333` is the 4:3 ratio of older photos and slides.

4 : 3

Source: Markdown

```
{% aspectratio ratio=1.3333 maw=320 %}
{% center h='100%' bg='var(--mantine-color-gray-2)' %}4 : 3{% endCenter %}
{% endAspectratio %}
```

Source: Python

```
inner = component('aardvark', 'center', h='100%',
                  bg='var(--mantine-color-gray-2)', children='4 : 3')
component('aardvark', 'aspectratio', ratio=1.3333, maw=320, children=inner)
```

With other components

The body is ordinary Markdown, so the most common use is wrapping a media embed — here a Markdown image — so the box holds its shape before the image arrives.

Source: Markdown

```
{% aspectratio ratio=1.7777 maw=480 %}
![A wide landscape placeholder](https://placehold.co/480x270?text=16:9)
{% endAspectratio %}
```

Source: Python

```
img = '![A wide landscape placeholder](https://placehold.co/480x270?text=16:9)'
component('aardvark', 'aspectratio', ratio=1.7777, maw=480, children=img)
```

Attributes

`ratio` is the one option of its own; everything else is the shared Mantine spacing/sizing system. Omit any attribute to take its Mantine default (ratio `1`).

Attribute	Values	Description
`ratio`	a number, width ÷ height (`1` default)	The aspect ratio. `1.7777` ≈ 16:9, `1.3333` ≈ 4:3, `1` = square.
`w` / `h`	`xs`–`xl` or any CSS size	Width / height of the box.

`miw` / `mih`	any CSS size	Minimum width / height.
`maw` / `mah`	any CSS size	Maximum width / height — `maw` caps an embed's width.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	`xs`–`xl` or any CSS size	Margin — all sides, or a single side / axis.
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	`xs`–`xl` or any CSS size	Padding — all sides, or a single side / axis.
`bg`	a theme color or any CSS color	Background color.
`c`	a theme color or any CSS color	Text (content) color.

CSS Selectors

Each `aspectratio` carries `data-aardvark-island="AspectRatio"` on its wrapper, and Mantine exposes its parts as `mantine-AspectRatio-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="AspectRatio"] {
  /* style every aspectratio on the page */
}

.mantine-AspectRatio-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Keeps a 16:9 box.

Source: Markdown

```
{% aspectratio ratio=1.7777 maw=480 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

```
Keeps a 16:9 box.
{% endAspectratio %}
```

Source: Python

```
component('aardvark', 'aspectratio', ratio=1.7777, maw=480,
          children='Keeps a 16:9 box.', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Box

`box` is the **base primitive** — the plain element every other Mantine component is built on. It draws nothing of its own; it just forwards Mantine's style props (spacing, color, sizing, raw `style`) onto a `<div>` so you can tweak padding, margin, a background, or sizing without pulling in a styled component. Because every Mantine component inherits this same style-prop surface, the `p`/`m`/`bg`/`c`/`w` props you learn here work on `paper`, `button`, `text`, and the rest too.

Use it as `{% box %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'box', …)`.

Demonstrations

The block body is the content; set any of the style props to shape it. Here a tinted background, padding, and a rounded corner via a raw `style`:

A plain box with a tinted background, padding, and rounded corners.

Source: Markdown

```
{% box bg='var(--mantine-color-blue-light)' p='md' style='border-radius: 8px' %}
A plain box with a tinted background, padding, and rounded corners.
{% endBox %}
```

Source: Python

```
component(
    'aardvark', 'box',
    bg='var(--mantine-color-blue-light)',
    p='md',
    style='border-radius: 8px',
    children='A plain box with a tinted background, padding, and rounded corners.',
)
```

The spacing and sizing props take a Mantine token (`xs`–`xl`) or any CSS length. Here large horizontal and small vertical padding, constrained to a max width:

Constrained to 320px, with large horizontal and small vertical padding.

Source: Markdown

```
{% box bg='var(--mantine-color-grape-light)' px='xl' py='sm' maw='320px' %}
Constrained to 320px, with large horizontal and small vertical padding.
{% endBox %}
```

Source: Python

```
component(
```

```
    'aardvark', 'box',
    bg='var(--mantine-color-grape-light)',
    px='xl', py='sm', maw='320px',
    children='Constrained to 320px, with large horizontal and small vertical padding.',
)
```

With other components

Box is a layout wrapper, so it composes with any other tag. Here it adds a background and padding around a `paper` surface and a `divider`:

A panel sitting inside a tinted, padded box.

Box just forwards the style props — the content inside is ordinary Markdown.

Source: Markdown

```
{% box bg='var(--mantine-color-gray-light)' p='lg' style='border-radius: 8px' %}
{% paper withBorder=true p='md' radius='md' %}
A panel sitting inside a tinted, padded box.
{% endPaper %}
{% divider my='md' %}
Box just forwards the style props — the content inside is ordinary Markdown.
{% endBox %}
```

Source: Python

```
inner = (
    component('aardvark', 'paper', withBorder=True, p='md', radius='md',
              children='A panel sitting inside a tinted, padded box.')
    + component('aardvark', 'divider', my='md')
    + 'Box just forwards the style props — the content inside is ordinary Markdown.'
)
component(
    'aardvark', 'box',
    bg='var(--mantine-color-gray-light)', p='lg',
    style='border-radius: 8px',
    children=inner,
)
```

Attributes

Box has no styling of its own — omit any attribute to leave that property unset. Spacing and sizing values take a Mantine token (`xs`–`xl`) or any CSS length.

Attribute	Valid values	Description

`bg`	A theme color (`blue.0`, `var(--mantine-color-blue-light)`) or any CSS color	Background color.
`c`	A theme color or any CSS color	Text color.
`opacity`	`0–1`	Element opacity.
`style`	A raw inline CSS string (`border-radius: 8px; color: red`)	Inline styles applied to the element.
`id`	Any string	HTML `id` on the rendered element, for CSS / JS selectors.
`m` / `mt` / `mb` / `ml` / `mr` / `mx` / `my`	Spacing token (`md`) or any CSS length	Margin — all sides, or a single side / axis.
`p` / `pt` / `pb` / `pl` / `pr` / `px` / `py`	Spacing token or any CSS length	Padding — all sides, or a single side / axis.
`w` / `h`	Any CSS length	Width / height.
`miw` / `mih` / `maw` / `mah`	Any CSS length	Min/max width and min/max height.

`attr={...}` forwards raw HTML attributes (e.g. a `data-*` hook) onto the rendered element.

CSS Selectors

Each `box` carries `data-aardvark-island="Box"` on its wrapper; it renders a single element with no Mantine Styles API parts, so target the island wrapper.

```
[data-aardvark-island="Box"] {
  /* style every box on the page */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

A plain box.

Source: Markdown

```
{% box bg='var(--mantine-color-blue-light)' p='md' style='border-radius: 8px'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
A plain box.
{% endBox %}
```

Source: Python

```
component(
    'aardvark', 'box',
    bg='var(--mantine-color-blue-light)',
    p='md',
    style='border-radius: 8px',
    children='A plain box.',
    attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
)
```

Center

`{% center %}` is a **built-in** tag that centers its content **both horizontally and vertically**. Give the container a height — with the `h` prop or from what's around it — and the body lands in the middle. It ships with Aardvark, so there's no setup.

Use it as `{% center %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'center', …)`.

Block centering

By default Center is a block-level flex container. Set a height so there's room to center within, and the body sits dead center.

[Centered]

Source: Markdown

```
{% center h=120 bg='var(--mantine-color-gray-1)' %}
{% badge size='lg' %}Centered{% endBadge %}
{% endCenter %}
```

Source: Python

```
inner = component('aardvark', 'badge', size='lg', children='Centered')
component('aardvark', 'center', h=120,
          bg='var(--mantine-color-gray-1)', children=inner)
```

Inline centering

`inline` switches the container to `inline-flex` — use it to vertically center an icon or badge alongside a run of text rather than as a block.

★ rated

Source: Markdown

```
{% center inline %}★ rated{% endCenter %}
```

Source: Python

```
component('aardvark', 'center', inline=True, children='★ rated')
```

With other components

The body is ordinary Markdown, so you can center a richer cluster — here a horizontal `{% group %}` of buttons inside a tall, tinted box.

Accept

Decline

Source: Markdown

```
{% center h=160 bg='var(--mantine-color-gray-1)' %}
{% group %}
{% button text='Accept' %}
{% button text='Decline' variant='outline' %}
{% endGroup %}
{% endCenter %}
```

Source: Python

```
inner = component('aardvark', 'group',
                  children=component('aardvark', 'button', text='Accept')
                           + component('aardvark', 'button', text='Decline',
variant='outline'))
component('aardvark', 'center', h=160,
          bg='var(--mantine-color-gray-1)', children=inner)
```

Attributes

`inline` is the one option of its own; everything else is the shared Mantine spacing/sizing system. `h` is the one you'll reach for most, since centering vertically needs a height.

Attribute	Values	Description
`inline`	`true`, `false` (default `false`)	Render as `inline-flex` (center inline content) instead of a block `flex`.
`w` / `h`	`xs`–`xl` or any CSS size	Width / height of the container. `h` gives room to center vertically.
`miw` / `mih`	any CSS size	Minimum width / height.
`maw` / `mah`	any CSS size	Maximum width / height.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	`xs`–`xl` or any CSS size	Margin — all sides, or a single side / axis.

`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	`xs`–`xl` or any CSS size	Padding — all sides, or a single side / axis.
`bg`	a theme color or any CSS color	Background color.
`c`	a theme color or any CSS color	Text (content) color.

CSS Selectors

Each `center` carries `data-aardvark-island="Center"` on its wrapper, and Mantine exposes its parts as `mantine-Center-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Center"] {
  /* style every center on the page */
}

.mantine-Center-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Centered both ways.

Source: Markdown

```
{% center h=120 bg='var(--mantine-color-gray-1)' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Centered both ways.
{% endCenter %}
```

Source: Python

```
component('aardvark', 'center', h=120,
          bg='var(--mantine-color-gray-1)', children='Centered both ways.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
```

```
'''})
```

Collapse

`collapse` is a **height-animation primitive**. It shows or hides its content with a smooth height transition, keyed off an open/closed flag (`opened`). On a static docs page the state is fixed at render time — `opened=true` renders the content visible, `opened=false` renders it collapsed away — so it's the lower-level building block underneath an interactive section, not a click-to-toggle widget by itself. For a ready-made collapsible section with no code, reach for the Accordion tag instead.

Use it as `{% collapse %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'collapse', …)`.

> **The static-page caveat.** Collapse animates content open and closed by height, which is inherently **stateful** — the animation runs when the open state changes. To get real interactive open/close, drive `opened` from your own component (a snippet that toggles it from React state, often paired with a button). Mantine's prop is named `in`, a reserved word in the template language, so this tag exposes it as **`opened`**.

Demonstrations

The block body is the collapsible content; `opened` fixes its state at render time. With `opened=true` the content is visible:

This content is inside a Collapse with `opened=true`, so it's shown. In a stateful component, toggling `opened` would animate it open and closed by height.

With `opened=false` the body is rendered collapsed — nothing visible below:

You won't see this on a static page — opened=false renders it collapsed.

Source: Markdown

```
{% collapse opened=true %}
Visible because opened=true. Set opened=false and it renders collapsed.
{% endCollapse %}

{% collapse opened=false %}
You won't see this on a static page — opened=false renders it collapsed.
{% endCollapse %}
```

Source: Python

```
component('aardvark', 'collapse', opened=True,
          children='Visible because opened=true.')
component('aardvark', 'collapse', opened=False,
          children="You won't see this on a static page.")
```

`transitionDuration` (ms), `transitionTimingFunction` (CSS easing), and `animateOpacity` tune the animation that runs when a stateful parent toggles `opened`:

A slower, eased expansion — these props shape the open/close animation a stateful parent triggers.

Source: Markdown

```
{% collapse opened=true transitionDuration=400 transitionTimingFunction='ease-in-out'
animateOpacity=true %}
A slower, eased expansion — these props shape the open/close animation a stateful parent
triggers.
{% endCollapse %}
```

Source: Python

```
component(
    'aardvark', 'collapse',
    opened=True,
    transitionDuration=400,
    transitionTimingFunction='ease-in-out',
    animateOpacity=True,
    children='A slower, eased expansion.',
)
```

With other components

Collapse is content-agnostic — the body is ordinary Markdown and other tags. Here a `paper` surface wraps an open collapse holding a `divider` and text:

Details

This panel's body is a Collapse — a stateful parent could hide it behind a toggle.

Source: Markdown

```
{% paper shadow='sm' withBorder=true radius='md' p='lg' %}
**Details**
{% collapse opened=true %}
{% divider my='sm' %}
This panel's body is a Collapse — a stateful parent could hide it behind a toggle.
{% endCollapse %}
{% endPaper %}
```

Source: Python

```
collapsed = (
    component('aardvark', 'divider', my='sm')
    + "This panel's body is a Collapse — a stateful parent could hide it behind a toggle."
)
inner = '**Details**' + component('aardvark', 'collapse', opened=True, children=collapsed)
```

```
component('aardvark', 'paper', shadow='sm', withBorder=True, radius='md', p='lg',
          children=inner)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`opened`	bare flag or `true` / `false` (default `true`)	Whether the content is shown. Maps to Mantine's `in`; `opened=false` collapses it.
`transitionDuration`	An integer (milliseconds)	Length of the open/close animation.
`transitionTimingFunction`	A CSS easing (`ease-in-out`, `linear`, `cubic-bezier(…)`)	Easing for the height animation.
`animateOpacity`	bare flag or `true` / `false` (default `true`)	Fade the content as it expands / collapses.

`attr={...}` forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `collapse` carries `data-aardvark-island="Collapse"` on its wrapper; it animates a single element with no Mantine Styles API parts, so target the island wrapper.

```
[data-aardvark-island="Collapse"] {
  /* style every collapse on the page */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Shown when opened.

Source: Markdown

```
{% collapse opened=true attr={'onclick': '''
const value = this.innerText;
```

```
console.log('attr demo value:', value);
alert(value);
'''} %}
Shown when opened.
{% endCollapse %}
```

Source: Python

```
component('aardvark', 'collapse', opened=True,
          children='Shown when opened.', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Container

`{% container %}` is a **built-in** tag that **centers content horizontally** and caps it to a comfortable reading width with a side gutter. It's the wrapper you reach for around a block of prose or a centered section. Set `size` to choose the max width, or `fluid` to take the full width while keeping the gutter.

Use it as `{% container %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'container', …)`.

Sized column

`size` is a Mantine size token (`xs`–`xl`) or any CSS width — it sets the **max content width**. Smaller tokens make a narrower column. A tinted background and padding make the bounds visible.

A narrow, centered column of content with a gutter on each side.

Source: Markdown

```
{% container size='xs' bg='var(--mantine-color-gray-1)' p='md' %}
A narrow, centered column of content with a gutter on each side.
{% endContainer %}
```

Source: Python

```
component('aardvark', 'container', size='xs',
          bg='var(--mantine-color-gray-1)', p='md',
          children='A narrow, centered column of content with a gutter on each side.')
```

A wider size

Bump `size` up to widen the column. `lg` is roomy enough for two-column content or a wide section.

A wider centered column — size='lg'.

Source: Markdown

```
{% container size='lg' bg='var(--mantine-color-gray-1)' p='md' %}
A wider centered column — size='lg'.
{% endContainer %}
```

Source: Python

```
component('aardvark', 'container', size='lg',
          bg='var(--mantine-color-gray-1)', p='md',
          children="A wider centered column — size='lg'.")
```

Fluid

`fluid` ignores `size` and takes the full available width, keeping only the gutter — handy for a full-bleed section that still wants side padding.

Full width, with a gutter.

Source: Markdown

```
{% container fluid bg='var(--mantine-color-gray-1)' p='md' %}
Full width, with a gutter.
{% endContainer %}
```

Source: Python

```
component('aardvark', 'container', fluid=True,
          bg='var(--mantine-color-gray-1)', p='md',
          children='Full width, with a gutter.')
```

With other components

The body is ordinary Markdown, so Container makes a tidy frame for a centered `{% stack %}` — a heading, a paragraph, and a call to action capped at a readable width.

[Welcome]

A short, readable intro paragraph that stays narrow enough to scan comfortably.

Get started

Source: Markdown

```
{% container size='sm' bg='var(--mantine-color-gray-1)' p='lg' %}
{% stack gap='sm' %}
{% badge size='lg' %}Welcome{% endBadge %}

A short, readable intro paragraph that stays narrow enough to scan comfortably.

{% button text='Get started' %}
{% endStack %}
{% endContainer %}
```

Source: Python

```
inner = component('aardvark', 'stack', gap='sm', children=(
    component('aardvark', 'badge', size='lg', children='Welcome')
    + '\n\nA short, readable intro paragraph that stays narrow enough to scan
```

```
comfortably.\n\n'
    + component('aardvark', 'button', text='Get started')))
component('aardvark', 'container', size='sm',
          bg='var(--mantine-color-gray-1)', p='lg', children=inner)
```

Attributes

`size` and `fluid` are the options of their own; everything else is the shared Mantine spacing/sizing system. Omit any attribute to take its Mantine default (size `md`).

Attribute	Values	Description
`size`	`xs`–`xl` or any CSS value (`md` default)	Max content width. Smaller tokens = narrower column.
`fluid`	`true`, `false` (default `false`)	Take the full width (ignoring `size`), keeping the gutter.
`w` / `h`	`xs`–`xl` or any CSS size	Width / height of the container.
`miw` / `mih`	any CSS size	Minimum width / height.
`maw` / `mah`	any CSS size	Maximum width / height.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	`xs`–`xl` or any CSS size	Margin — all sides, or a single side / axis.
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	`xs`–`xl` or any CSS size	Padding — all sides, or a single side / axis.
`bg`	a theme color or any CSS color	Background color.
`c`	a theme color or any CSS color	Text (content) color.

CSS Selectors

Each `container` carries `data-aardvark-island="Container"` on its wrapper, and Mantine exposes its parts as `mantine-Container-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Container"] {
  /* style every container on the page */
}
```

```
.mantine-Container-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Centered, max-width content.

Source: Markdown

```
{% container size='xs' bg='var(--mantine-color-gray-1)' p='md' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Centered, max-width content.
{% endContainer %}
```

Source: Python

```
component('aardvark', 'container', size='xs',
          bg='var(--mantine-color-gray-1)', p='md',
          children='Centered, max-width content.', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Flex

`{% flex %}` is a **built-in** tag for a flexbox container with the **full CSS flex surface** as props. Where `{% group %}` and `{% stack %}` are opinionated row/column shortcuts, `flex` gives you `direction`, `wrap`, `justify`, `align`, and per-axis gaps directly. Reach for it when you need precise control over the flex layout.

Use it as `{% flex %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'flex', …)`.

Justify and align

`justify` is the CSS `justify-content` and `align` is `align-items`. Together they place the children along the main and cross axes — here pushed to the edges and vertically centered.

[Left] [Right]

Source: Markdown

```
{% flex justify='space-between' align='center' gap='md' %}
{% badge %}Left{% endBadge %} {% badge %}Right{% endBadge %}
{% endFlex %}
```

Source: Python

```
inner = (component('aardvark', 'badge', children='Left')
         + ' '
         + component('aardvark', 'badge', children='Right'))
component('aardvark', 'flex', justify='space-between',
          align='center', gap='md', children=inner)
```

Direction

`direction` is any CSS `flex-direction` — `row` (the default), `column`, `row-reverse`, or `column-reverse`. A `column` direction stacks the children top to bottom.

First

Second

Source: Markdown

```
{% flex direction='column' gap='sm' %}
{% button text='First' %} {% button text='Second' %}
{% endFlex %}
```

Source: Python

```
inner = (component('aardvark', 'button', text='First')
         + ' '
         + component('aardvark', 'button', text='Second'))
component('aardvark', 'flex', direction='column', gap='sm', children=inner)
```

Wrap and per-axis gaps

`wrap` is any CSS `flex-wrap` (`nowrap`, `wrap`, `wrap-reverse`); when children wrap onto a new line, `rowGap` and `columnGap` set the gaps independently — vertical and horizontal.

[One] [Two] [Three] [Four] [Five]

Source: Markdown

```
{% flex wrap='wrap' rowGap='lg' columnGap='xs' maw=260 %}
{% badge %}One{% endBadge %} {% badge %}Two{% endBadge %} {% badge %}Three{% endBadge %} {%
badge %}Four{% endBadge %} {% badge %}Five{% endBadge %}
{% endFlex %}
```

Source: Python

```
inner = ' '.join(
    component('aardvark', 'badge', children=t)
    for t in ('One', 'Two', 'Three', 'Four', 'Five'))
component('aardvark', 'flex', wrap='wrap', rowGap='lg',
          columnGap='xs', maw=260, children=inner)
```

With other components

The body is ordinary Markdown, so Flex makes a clean toolbar — a label pushed left, actions pushed right.

[Filters]

Reset

Apply

Source: Markdown

```
{% flex justify='space-between' align='center' gap='md' bg='var(--mantine-color-gray-1)'
p='sm' %}
{% badge size='lg' %}Filters{% endBadge %}
{% group gap='xs' %}
{% button text='Reset' variant='subtle' %}
{% button text='Apply' %}
{% endGroup %}
{% endFlex %}
```

Source: Python

```
actions = component('aardvark', 'group', gap='xs', children=(
    component('aardvark', 'button', text='Reset', variant='subtle')
    + component('aardvark', 'button', text='Apply')))
inner = component('aardvark', 'badge', size='lg', children='Filters') + actions
component('aardvark', 'flex', justify='space-between', align='center',
          gap='md', bg='var(--mantine-color-gray-1)', p='sm', children=inner)
```

Attributes

Flex sets no defaults of its own — it's a thin Box over CSS flex, so an omitted option falls back to the CSS initial value. The flex options come first; the rest is the shared Mantine spacing/sizing system.

Attribute	Values	Description
`direction`	`row`, `column`, `row-reverse`, `column-reverse`	CSS `flex-direction`.
`wrap`	`nowrap`, `wrap`, `wrap-reverse`	CSS `flex-wrap`.
`justify`	any CSS `justify-content` (`flex-start`, `center`, `space-between`, …)	Placement along the main axis.
`align`	any CSS `align-items` (`stretch`, `center`, `flex-start`, …)	Placement along the cross axis.
`gap`	`xs`–`xl` or any CSS value	Gap on both axes.
`rowGap`	`xs`–`xl` or any CSS value	Vertical gap only.
`columnGap`	`xs`–`xl` or any CSS value	Horizontal gap only.
`w` / `h`	`xs`–`xl` or any CSS size	Width / height of the container.
`miw` / `mih`	any CSS size	Minimum width / height.
`maw` / `mah`	any CSS size	Maximum width / height.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	`xs`–`xl` or any CSS size	Margin — all sides, or a single side / axis.

`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	`xs`–`xl` or any CSS size	Padding — all sides, or a single side / axis.
`bg`	a theme color or any CSS color	Background color.
`c`	a theme color or any CSS color	Text (content) color.

CSS Selectors

Each `flex` carries `data-aardvark-island="Flex"` on its wrapper, and Mantine exposes its parts as `mantine-Flex-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Flex"] {
  /* style every flex on the page */
}

.mantine-Flex-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Left Right

Source: Markdown

```
{% flex justify='space-between' align='center' gap='md' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Left {% space w='md' %} Right
{% endFlex %}
```

Source: Python

```
inner = 'Left ' + component('aardvark', 'space', w='md') + ' Right'
print(component('aardvark', 'flex', justify='space-between', align='center',
          gap='md', children=inner, attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}))
```

Grid

`{% grid %}` is a **built-in** tag for a flexible **12-column grid** where cells can span a chosen number of columns. Use it when you want columns of different widths — a sidebar next to a wide main area, say. For an equal-width grid, reach for `{% simplegrid %}` instead.

Use it as `{% grid %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'grid', …)`.

Cells with `items`

The simplest way to lay out cells of different widths is the `items` param — a `|`-separated list of cell texts — paired with `spans`, a comma-separated list of column spans (out of 12) for each cell. This form is self-closing (no block body).

Source: Markdown

```
{% grid items='Sidebar|Main content area' spans='3,9' %}
```

Source: Python

```
component('aardvark', 'grid', items='Sidebar|Main content area', spans='3,9')
```

Auto spans

A `span` of `auto` lets a cell take whatever width is left. Mix a fixed span with `auto` cells to pin one column and let the rest share the remainder.

Source: Markdown

```
{% grid items='Fixed 4|Auto|Auto' spans='4,auto,auto' %}
```

Source: Python

```
component('aardvark', 'grid', items='Fixed 4|Auto|Auto', spans='4,auto,auto')
```

Cells from the block body

Without `items`, each top-level block in the body becomes one cell — handy when a cell holds richer Markdown than a plain string. Set `span` to give every body cell the same column span, and `gap` to widen the gap between cells.

[Half] [Half]

Source: Markdown

```
{% grid span=6 gap='lg' %}
{% badge fullWidth size='lg' %}Half{% endBadge %}

{% badge fullWidth size='lg' %}Half{% endBadge %}
{% endGrid %}
```

Source: Python

```
cell = component('aardvark', 'badge', fullWidth=True, size='lg', children='Half')
component('aardvark', 'grid', span='6', gap='lg',
          children=cell + '\n\n' + cell)
```

Grow and custom column count

`grow` lets the last row's cells grow to fill the remaining width. `columns` changes the total number of tracks — here a 6-column grid with two 2-span cells that grow to fill.

Source: Markdown

```
{% grid items='A|B' spans='2,2' columns=6 grow=true %}
```

Source: Python

```
component('aardvark', 'grid', items='A|B', spans='2,2', columns=6, grow=True)
```

With other components

Body cells are ordinary Markdown, so a grid is the natural frame for a dashboard row of rich cells. Each top-level block becomes one cell; set the grid-wide `span` to size them — here two half-width stat cards (`span=6` of 12).

Revenue

A panel spanning half the row.

Today

A second panel beside it.

Source: Markdown

```
{% grid span=6 gap='md' %}
{% card title='Revenue' %}A panel spanning half the row.{% endCard %}

{% card title='Today' %}A second panel beside it.{% endCard %}
{% endGrid %}
```

Source: Python

```
left = component('aardvark', 'card', title='Revenue',
                 children='A panel spanning half the row.')
right = component('aardvark', 'card', title='Today',
                  children='A second panel beside it.')
component('aardvark', 'grid', span='6', gap='md',
          children=left + '\n\n' + right)
```

Attributes

Omit any attribute to take its Mantine default (columns `12`, gap `md`, span `auto`).

Attribute	Values	Description
`items`	a `|`-separated string of cell texts	Cells as a param (alternative to the block body).
`spans`	comma-separated, one per `items` cell — a number 1–12, `auto`, or `content`	Column span for each `items` cell.
`span`	a number 1–12, `auto`, or `content`	Default column span for every cell (used when a cell has no `spans` entry).
`columns`	a number (`12` default)	Total number of grid columns.
`gap`	`xs`–`xl` or any CSS value (`md` default)	Gap between cells.
`justify`	any CSS `justify-content`	Placement of the row of cells along the main axis.
`align`	any CSS `align-items`	Placement of the cells along the cross axis.
`grow`	`true`, `false` (default `false`)	Let the last row's cells grow to fill the remaining width.
`overflow`	`hidden`, `visible` (`hidden` default)	`overflow` on the grid root. Set `visible` to let content spill.

CSS Selectors

Each `grid` carries `data-aardvark-island="Grid"` on its wrapper, and Mantine exposes its parts as `mantine-Grid-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Grid"] {
  /* style every grid on the page */
}

.mantine-Grid-root {
  /* the root part */
}

.mantine-Grid-inner {
  /* the inner part */
}

.mantine-Grid-col {
  /* the col part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Source: Markdown

```
{% grid items='Sidebar|Main content area' spans='3,9' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'grid', items='Sidebar|Main content area', spans='3,9',
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Group

`{% group %}` lays its children out in a **horizontal row** with a consistent gap — the everyday way to put buttons, badges, or chips side by side. It is a flex row: control where children sit on the main axis (`justify`), how they line up on the cross axis (`align`), the space between them (`gap`), whether they wrap (`wrap`), and whether they share the row width equally (`grow`). The block body is the content.

Use it as `{% group %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'group', …)`.

Basic row

With no options, children sit at the start of the row with the default `md` gap.

[One] [Two] [Three]

Source: Markdown

```
{% group %}
{% badge %}One{% endBadge %} {% badge %}Two{% endBadge %} {% badge %}Three{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', children=(
    component('aardvark', 'badge', children='One')
    + component('aardvark', 'badge', children='Two')
    + component('aardvark', 'badge', children='Three')
))
```

Justify (main axis)

`justify` sets `justify-content` — where the children sit along the row. Common values are `flex-start` (default), `center`, `flex-end`, `space-between`, and `space-around`.

[One] [Two] [Three]

Source: Markdown

```
{% group justify='center' %}
{% badge %}One{% endBadge %} {% badge %}Two{% endBadge %} {% badge %}Three{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', justify='center', children=(
```

```
    component('aardvark', 'badge', children='One')
    + component('aardvark', 'badge', children='Two')
    + component('aardvark', 'badge', children='Three')
))
```

Align (cross axis)

`align` sets `align-items` — how children line up across the row when they differ in height. Values include `center` (default), `flex-start`, `flex-end`, and `stretch`.

[xs] [lg] [xl]

Source: Markdown

```
{% group align='flex-end' %}
{% badge size='xs' %}xs{% endBadge %} {% badge size='lg' %}lg{% endBadge %} {% badge
size='xl' %}xl{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', align='flex-end', children=(
    component('aardvark', 'badge', size='xs', children='xs')
    + component('aardvark', 'badge', size='lg', children='lg')
    + component('aardvark', 'badge', size='xl', children='xl')
))
```

Gap

`gap` is the space between children — a Mantine size token (`xs`–`xl`) or any CSS value (`8px`, `2rem`). The default is `md`.

[One] [Two] [Three]

Source: Markdown

```
{% group gap='xl' %}
{% badge %}One{% endBadge %} {% badge %}Two{% endBadge %} {% badge %}Three{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', gap='xl', children=(
    component('aardvark', 'badge', children='One')
    + component('aardvark', 'badge', children='Two')
    + component('aardvark', 'badge', children='Three')
))
```

Wrap

`wrap` sets `flex-wrap`. The default is `wrap` (children flow onto the next line when they run out of room); set `wrap='nowrap'` to force everything onto a single line.

[One] [Two] [Three] [Four]

Source: Markdown

```
{% group wrap='nowrap' %}
{% badge %}One{% endBadge %} {% badge %}Two{% endBadge %} {% badge %}Three{% endBadge %} {%
badge %}Four{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', wrap='nowrap', children=(
    component('aardvark', 'badge', children='One')
    + component('aardvark', 'badge', children='Two')
    + component('aardvark', 'badge', children='Three')
    + component('aardvark', 'badge', children='Four')
))
```

Grow

`grow` (a bare flag) makes each child share the row width equally. By default `preventGrowOverflow` is on, so each child reserves an equal slice and a wide child can't push the others around; set `preventGrowOverflow='false'` to let children size to their content.

Left

Middle

Right

Source: Markdown

```
{% group grow %}
{% button text='Left' %} {% button text='Middle' %} {% button text='Right' %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', grow=True, children=(
    component('aardvark', 'button', text='Left')
    + component('aardvark', 'button', text='Middle')
    + component('aardvark', 'button', text='Right')
```

```
))
```

With other components

`group` composes with any other tag. Here it puts a `button` and two `badge`s on one line, pushed apart with `justify='space-between'`.

Deploy

[Ready] [v2]

Source: Markdown

```
{% group justify='space-between' align='center' %}
{% button text='Deploy' %}
{% group gap='xs' %}
{% badge color='green' %}Ready{% endBadge %} {% badge color='blue' variant='light' %}v2{%
endBadge %}
{% endGroup %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', justify='space-between', align='center', children=(
    component('aardvark', 'button', text='Deploy')
    + component('aardvark', 'group', gap='xs', children=(
        component('aardvark', 'badge', color='green', children='Ready')
        + component('aardvark', 'badge', color='blue', variant='light', children='v2')
    ))
))
```

Attributes

Attribute	Valid values	Description
`justify`	`flex-start` (default), `center`, `flex-end`, `space-between`, `space-around`, `space-evenly`	`justify-content` for the row — where children sit on the main axis.
`align`	`center` (default), `flex-start`, `flex-end`, `stretch`, `baseline`	`align-items` — how children line up on the cross axis.
`gap`	`xs`, `sm`, `md` (default), `lg`, `xl`, or any CSS value	Space between children.
`wrap`	`wrap` (default), `nowrap`, `wrap-reverse`	`flex-wrap` for the row.
`grow`	bare flag (off by default)	Make children share the row width equally.
`preventGrowOverflow`	`true` (default), `false`	With `grow`, reserve equal width so a wide child can't push others; set `false` to let children size to content.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	Mantine size token or any CSS value	Margin (all sides / top / bottom / left / right / horizontal / vertical).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	Mantine size token or any CSS value	Padding (all sides / top / bottom / left / right / horizontal / vertical).
`bg`	color name or CSS color	Background color.
`c`	color name or CSS color	Text color.
`w`, `h`	Mantine size token or any CSS value	Width / height.
`miw`, `mih`, `maw`, `mah`	any CSS value	Min/max width and min/max height.

CSS Selectors

Each `group` carries `data-aardvark-island="Group"` on its wrapper, and Mantine exposes its parts as `mantine-Group-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Group"] {
```

```
  /* style every group on the page */
}

.mantine-Group-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

[one] [two]

Source: Markdown

```
{% group attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% badge %}one{% endBadge %} {% badge %}two{% endBadge %}
{% endGroup %}
```

Source: Python

```
inner = (component('aardvark', 'badge', children='one')
         + ' '
         + component('aardvark', 'badge', children='two'))
print(component('aardvark', 'group', children=inner, attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}))
```

Marquee

`marquee` is a **scrolling ticker** — content that slides continuously across the row, the way logo strips and announcement bars do. The content is repeated so the loop is seamless, it pauses while the pointer is over it (`pauseOnHover`), and it respects `prefers-reduced-motion` (the content holds still for readers who've asked to reduce motion).

Use it as `{% marquee %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'marquee', …)`.

Demonstrations

The block body is the scrolling content; `duration` sets how long one loop takes in milliseconds (lower is faster):

Ship docs faster • Markdown in, static site out • no build step to babysit •

Source: Markdown

```
{% marquee duration=15000 %}
Ship docs faster — Markdown in, static site out — no build step to babysit —  
{% endMarquee %}
```

Source: Python

```
component(
    'aardvark', 'marquee',
    duration=15000,
    children='Ship docs faster — Markdown in, static site out — no build step to babysit —
 ',
)
```

`direction='right'` reverses the scroll and a lower `duration` moves it faster. The items can themselves be other built-in tags:

[React] [Mantine] [Markdown] [Static]

Source: Markdown

```
{% marquee direction='right' duration=8000 %}
{% badge color='blue' %}React{% endBadge %} {% badge color='grape' %}Mantine{% endBadge %}
{% badge color='green' %}Markdown{% endBadge %} {% badge color='orange' %}Static{% endBadge
%}  
{% endMarquee %}
```

Source: Python

```
badges = ' '.join(
```

```
    component('aardvark', 'badge', color=c, children=label)
    for c, label in (('blue', 'React'), ('grape', 'Mantine'),
                     ('green', 'Markdown'), ('orange', 'Static'))
)
component('aardvark', 'marquee', direction='right', duration=8000,
          children=badges + '  ')
```

`gap` widens the space between the looping copies, and `pauseOnHover=false` keeps it scrolling even under the pointer:

Never stops • wider gap between repeats •

Source: Markdown

```
{% marquee gap='4rem' pauseOnHover=false %}
Never stops — wider gap between repeats —  
{% endMarquee %}
```

Source: Python

```
component(
    'aardvark', 'marquee',
    gap='4rem', pauseOnHover=False,
    children='Never stops — wider gap between repeats —  ',
)
```

By default the strip pauses while the pointer is over it (`pauseOnHover` is on) — hover any marquee above to see it hold.

With other components

A marquee makes a tidy announcement bar inside a `paper` surface:

[live] Maintenance window tonight 22:00 UTC • status page updated •

Source: Markdown

```
{% paper withBorder=true radius='md' p='sm' %}
{% marquee %}
{% badge color='red' %}live{% endBadge %}   Maintenance window tonight 22:00 UTC
 •  status page updated  • 
{% endMarquee %}
{% endPaper %}
```

Source: Python

```
ticker = component(
    'aardvark', 'marquee',
    children=(
        component('aardvark', 'badge', color='red', children='live')
```

```
        + '   Maintenance window tonight 22:00 UTC  ·  status page updated
 · '
    ),
)
component('aardvark', 'paper', withBorder=True, radius='md', p='sm', children=ticker)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`direction`	`left` (default) / `right` / `up` / `down`	Scroll direction (`up`/`down` scroll vertically).
`duration`	An integer, milliseconds for one loop (default `40000`)	Loop length — lower is faster.
`gap`	Any CSS length (default `2rem`)	Space between the looping copies.
`pauseOnHover`	bare flag or `true` / `false` (default `true`)	Pause the scroll while the pointer is over it. Set `false` to keep scrolling.
`fadeEdges`	`true` / `false` (default `false`)	Fade the content to transparent at the start/end edges.

`attr={...}` forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `marquee` carries `data-aardvark-island="Marquee"` on its wrapper, and Mantine exposes its parts as `mantine-Marquee-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Marquee"] {
  /* style every marquee on the page */
}

.mantine-Marquee-root {
  /* the root part */
}

.mantine-Marquee-group {
  /* the group part */
}

.mantine-Marquee-content {
  /* the content part */
```

```
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

This text scrolls across the screen.

Source: Markdown

```
{% marquee duration=15000 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
This text scrolls across the screen.
{% endMarquee %}
```

Source: Python

```
component('aardvark', 'marquee', duration=15000,
          children='This text scrolls across the screen.', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Paper

`paper` is a **surface** — a panel that lifts content off the page with a shadow, rounded corners, padding, and an optional border. It ships with Aardvark, so a surface is a single tag with no setup. The block body is the panel content, and the tag adds a comfortable padding by default.

Use it as `{% paper %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'paper', …)`.

Demonstrations

A surface with a shadow, rounded corners, a border, and roomy padding. Any Markdown goes inside — text, lists, even other tags:

A panel. Drop any Markdown in here — text, lists, even other tags.

Source: Markdown

```
{% paper shadow='md' radius='md' withBorder=true p='lg' %}
**A panel.** Drop any Markdown in here — text, lists, even other tags.
{% endPaper %}
```

Source: Python

```
component(
    'aardvark', 'paper',
    shadow='md', radius='md', withBorder=True, p='lg',
    children='**A panel.** Drop any Markdown in here — text, lists, even other tags.',
)
```

The `shadow` prop controls elevation — `xs` is subtle, `xl` floats:

shadow xs, with border

shadow md

shadow xl

Source: Markdown

```
{% paper shadow='xs' p='md' withBorder=true %}shadow xs, with border{% endPaper %}
{% paper shadow='md' p='md' %}shadow md{% endPaper %}
{% paper shadow='xl' p='md' %}shadow xl{% endPaper %}
```

Source: Python

```
for sh in ('xs', 'md', 'xl'):
    component('aardvark', 'paper', shadow=sh, p='md', children=f'shadow {sh}')
```

`withBorder` adds a 1px outline and `radius` rounds the corners — here with extra padding:

A bordered card with large corner rounding and extra padding.

Source: Markdown

```
{% paper withBorder=true radius='lg' p='xl' %}
A bordered card with large corner rounding and extra padding.
{% endPaper %}
```

Source: Python

```
component(
    'aardvark', 'paper',
    withBorder=True, radius='lg', p='xl',
    children='A bordered card with large corner rounding and extra padding.',
)
```

With other components

A surface is a natural container for other tags. Here a panel holds a `divider` and a `badge`:

Release notes

Version 2.0 is out [new]

Source: Markdown

```
{% paper shadow='sm' withBorder=true radius='md' p='lg' %}
**Release notes**
{% divider my='sm' %}
Version 2.0 is out {% badge color='green' %}new{% endBadge %}
{% endPaper %}
```

Source: Python

```
inner = (
    '**Release notes**'
    + component('aardvark', 'divider', my='sm')
    + 'Version 2.0 is out '
    + component('aardvark', 'badge', color='green', children='new')
)
component(
    'aardvark', 'paper',
    shadow='sm', withBorder=True, radius='md', p='lg',
    children=inner,
)
```

Attributes

Omit any attribute to take its Mantine default (no shadow, no border). Padding defaults to `md`.

Attribute	Valid values	Description
`shadow`	`xs` / `sm` / `md` / `lg` / `xl`	Drop shadow / elevation. Omit for no shadow.
`radius`	`xs`–`xl`, or any CSS value	Corner rounding.
`withBorder`	bare flag or `true` / `false` (default `false`)	Add a 1px border.
`p` / `pt` / `pb` / `pl` / `pr` / `px` / `py`	Spacing token (`md`) or any CSS value	Padding — all sides, a side, or an axis. Defaults to `md`.
`m` / `mt` / `mb` / `ml` / `mr` / `mx` / `my`	Spacing token or any CSS value	Margin.
`bg`	A theme color or any CSS color	Background color.
`c`	A theme color or any CSS color	Text color.
`w` / `h`	Any CSS length	Width / height.
`miw` / `mih` / `maw` / `mah`	Any CSS length	Min/max width and min/max height.

`attr={...}` forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `paper` carries `data-aardvark-island="Paper"` on its wrapper, and Mantine exposes its parts as `mantine-Paper-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Paper"] {
  /* style every paper on the page */
}

.mantine-Paper-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

A panel.

Source: Markdown

```
{% paper shadow='md' radius='md' withBorder=true p='lg' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
**A panel.**
{% endPaper %}
```

Source: Python

```
component(
    'aardvark', 'paper',
    shadow='md', radius='md', withBorder=True, p='lg',
    children='**A panel.**',
    attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
)
```

Portal

`portal` is a low-level **render-target primitive**: it renders its children into a different part of the DOM — by default the end of `document.body` — instead of where the tag sits in the page. That lets overlays, popovers, and tooltips escape a parent's `overflow: hidden` or its stacking context (`z-index`) so they're never clipped. It has no visual styling of its own, so on a static docs page the portalled content simply appears at the bottom of the page rather than inline — that's the primitive working as intended, not a bug. You'll mostly meet it indirectly, inside Modal, Drawer, and Tooltip.

Use it as `{% portal %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'portal', …)`.

Demonstrations

The block body is the content that gets portalled out. By default it mounts at the end of `document.body`, so on this page it renders below the normal content flow rather than at this exact spot.

Preview (the portalled text appears at the bottom of the page, not here):

This paragraph was portalled into document.body by the Portal docs page.

Source: Markdown

```
{% portal %}
This paragraph is rendered into document.body, not here inline.
{% endPortal %}
```

Source: Python

```
component('aardvark', 'portal',
          children='This paragraph is rendered into document.body, not here inline.')
```

Targeting a specific node

Pass `target` (a CSS selector or element id) to mount into a specific element instead of `document.body`. This is the same mechanism Mantine's overlay components use under the hood; reach for the tag directly only when you're building a custom overlay yourself.

Preview — the body mounts into `#my-overlay-root` if that element exists, otherwise into `document.body`:

Source: Markdown

```
{% portal target='#my-overlay-root' %}
…content rendered into #my-overlay-root…
{% endPortal %}
```

Source: Python

```
component('aardvark', 'portal',
          target='#my-overlay-root',
          children='…content rendered into #my-overlay-root…')
```

Reusing a single target node

Set `reuseTargetNode=true` so many portals share one generated target node instead of each creating its own — handy when you portal a lot of small fragments and don't want a node per portal.

Source: Markdown

```
{% portal reuseTargetNode=true %}
…content sharing a single reused target node…
{% endPortal %}
```

Source: Python

```
component('aardvark', 'portal',
          reuseTargetNode=True,
          children='…content sharing a single reused target node…')
```

With other components

Portal is most useful wrapping content that must escape a clipped container. Here a Paper surface is portalled to `document.body` so it can't be cut off by an ancestor's `overflow: hidden`.

Preview (the Paper renders at the bottom of the page):

This Paper was portalled to document.body so it escapes any clipping ancestor.

Source: Markdown

```
{% portal %}
{% paper withBorder=true p='md' radius='md' %}This Paper was portalled to document.body so
it escapes any clipping ancestor.{% endPaper %}
{% endPortal %}
```

Source: Python

```
component('aardvark', 'portal',
          children=component('aardvark', 'paper',
                             withBorder=True, p='md', radius='md',
                             children='This Paper was portalled to document.body.'))
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `reuseTargetNode`) become `=True`.

Attribute	Valid values	Description
`target`	A CSS selector or element id (string)	The element to render into, instead of `document.body`.
`reuseTargetNode`	`true` / `false` (default `false`)	Reuse a single shared target node across portals, rather than creating one per portal.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `portal` carries `data-aardvark-island="Portal"` on its wrapper; it renders its content into `document.body` with no wrapper element, so target the island wrapper.

```
[data-aardvark-island="Portal"] {
  /* style every portal on the page */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

This paragraph is portalled into document.body.

Source: Markdown

```
{% portal attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
This paragraph is portalled into document.body.
{% endPortal %}
```

Source: Python

```
component('aardvark', 'portal',
          children='This paragraph is portalled into document.body.',
          attr={'onclick': '''
const value = this.innerText;
```

```
console.log('attr demo value:', value);
alert(value);
'''})
```

Scroll Buttons

`scrollbuttons` is a **horizontal scroller with prev/next buttons**. The block body is a wide row of content that scrolls sideways, and chevron controls fade in at each edge so a reader can page through it a screenful at a time. The controls only show when there is more content to reach in that direction, and the row can also be dragged with the pointer. It is the right tool for a strip of cards, a row of images, or any rail too wide for the page.

This is a different widget from `scroller`, which is a plain native-overflow box with the browser's own scrollbars — `scrollbuttons` adds the paging chevron controls.

Use it as `{% scrollbuttons %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'scrollbuttons', …)`.

Demonstrations

The block body is the scrolling row; give the children enough width to overflow so the controls have something to page to. `scrollAmount` sets how many pixels each chevron click moves:

First card

Second card

Third card

Fourth card

Fifth card

Sixth card

Source: Markdown

```
{% scrollbuttons scrollAmount=240 %}
{% group wrap='nowrap' gap='md' %}
{% paper withBorder=true radius='md' p='lg' w='200' %}First card{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}Second card{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}Third card{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}Fourth card{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}Fifth card{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}Sixth card{% endPaper %}
{% endGroup %}
{% endScrollbuttons %}
```

Source: Python

```
cards = ''.join(
    component('aardvark', 'paper', withBorder=True, radius='md', p='lg', w='200',
              children=label)
    for label in ('First card', 'Second card', 'Third card',
```

```
                  'Fourth card', 'Fifth card', 'Sixth card')
)
row = component('aardvark', 'group', wrap='nowrap', gap='md', children=cards)
component('aardvark', 'scrollbuttons', scrollAmount=240, children=row)
```

Pin both controls on with `showStartControl` / `showEndControl` (so they show even at the ends), and `controlSize` resizes the chevron buttons:

[React] [Mantine] [Markdown] [Static] [Islands] [Themes] [Search] [Analytics]

Source: Markdown

```
{% scrollbuttons showStartControl=true showEndControl=true controlSize='48' %}
{% group wrap='nowrap' gap='sm' %}
{% badge size='xl' color='blue' %}React{% endBadge %}
{% badge size='xl' color='grape' %}Mantine{% endBadge %}
{% badge size='xl' color='green' %}Markdown{% endBadge %}
{% badge size='xl' color='orange' %}Static{% endBadge %}
{% badge size='xl' color='red' %}Islands{% endBadge %}
{% badge size='xl' color='teal' %}Themes{% endBadge %}
{% badge size='xl' color='indigo' %}Search{% endBadge %}
{% badge size='xl' color='pink' %}Analytics{% endBadge %}
{% endGroup %}
{% endScrollbuttons %}
```

Source: Python

```
badges = ''.join(
    component('aardvark', 'badge', size='xl', color=c, children=label)
    for c, label in (('blue', 'React'), ('grape', 'Mantine'), ('green', 'Markdown'),
                     ('orange', 'Static'), ('red', 'Islands'), ('teal', 'Themes'),
                     ('indigo', 'Search'), ('pink', 'Analytics'))
)
row = component('aardvark', 'group', wrap='nowrap', gap='sm', children=badges)
component('aardvark', 'scrollbuttons',
          showStartControl=True, showEndControl=True, controlSize='48', children=row)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`scrollAmount`	An integer, pixels (default `200`)	How far each chevron click scrolls the row.
`controlSize`	A number or CSS length (default `60px`)	Size of the chevron control buttons.

`edgeGradientColor`	Any CSS color (default the page background)	Color of the fade behind the controls.
`draggable`	bare flag or `true` / `false` (default `true`)	Allow dragging the row with the pointer. Set `false` to disable.
`showStartControl`	`true` / `false` (default `false`)	Keep the start (left) control visible even at that end.
`showEndControl`	`true` / `false` (default `false`)	Keep the end (right) control visible even at that end.

`attr={...}` forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `scrollbuttons` mounts as an island, so its outer wrapper carries `data-aardvark-island="Scroller"` — a stable hook independent of Mantine's hashed class names. Inside, the native Scroller exposes its Styles API parts as `mantine-Scroller-*` classes: `root` is the component, `container` the clipping frame, `content` the scrolling track that holds the children, `control` each chevron button, and `chevron` the icon within a control.

```
/* The island wrapper — the most stable selector. */
[data-aardvark-island="Scroller"] {
  margin-block: 1rem;
}

/* Mantine Styles API parts. */
.mantine-Scroller-root {}
.mantine-Scroller-container {}
.mantine-Scroller-content {}      /* the scrolling track of children */
.mantine-Scroller-control {       /* the prev / next chevron buttons */
  background: var(--mantine-color-body);
}
.mantine-Scroller-chevron {}      /* the icon inside a control */
```

Injecting Attributes

`attr={...}` passes raw HTML attributes straight onto the rendered element — handy for ids, data hooks, ARIA, or inline handlers the macro doesn't expose as named options. Pass a dictionary literal; every key/value lands verbatim on the element.

Below, an injected `onclick` reports the element's text content when the row is clicked:

Click anywhere on this row

to see its text content

via the injected handler

then keep scrolling

Source: Markdown

```
{% scrollbuttons attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% group wrap='nowrap' gap='md' %}
{% paper withBorder=true radius='md' p='lg' w='200' %}Click anywhere on this row{% endPaper
%}
{% paper withBorder=true radius='md' p='lg' w='200' %}to see its text content{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}via the injected handler{% endPaper %}
{% paper withBorder=true radius='md' p='lg' w='200' %}then keep scrolling{% endPaper %}
{% endGroup %}
{% endScrollbuttons %}
```

Source: Python

```
cards = ''.join(
    component('aardvark', 'paper', withBorder=True, radius='md', p='lg', w='200',
              children=label)
    for label in ('Click anywhere on this row', 'to see its text content',
                  'via the injected handler', 'then keep scrolling')
)
row = component('aardvark', 'group', wrap='nowrap', gap='md', children=cards)
print(component('aardvark', 'scrollbuttons', children=row, attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}))
```

ScrollArea

`scrollarea` is a **scrollable region** with styled, auto-hiding overlay scrollbars — nicer than the browser's default bars, and consistent across macOS, Windows, and Linux. Give it a fixed height with `h` (and/or width with `w`) so its content can overflow and scroll. Use `type` to control when the bars appear and `offsetScrollbars` to inset the content so the bars never overlap it. Close the block with `{% endScrollarea %}` (one capital S).

Use it as `{% scrollarea %}…{% endScrollarea %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'scrollarea', …)`.

Demonstrations

Constrain the height with `h`; the block body is the (overflowing) content. `type='always'` keeps the styled bar visible.

Preview:

A tall block of content that overflows the 160px-tall area and scrolls.

Mantine's ScrollArea draws its own overlay scrollbar instead of the browser's, so it looks the same on macOS, Windows, and Linux.

Keep scrolling — there's more here than fits in 160 pixels of height, which is the whole point of constraining the area and letting the body overflow.

The bar tracks your position and (with `type='hover'`) fades out when you're not interacting.

Source: Markdown

```
{% scrollarea h='160' type='always' %}
A tall block of content that overflows the 160px-tall area and scrolls.
Add enough paragraphs here and the styled scrollbar appears on the right.
{% endScrollarea %}
```

Source: Python

```
component('aardvark', 'scrollarea',
          h='160', type='always',
          children='A tall block of content that overflows the area and scrolls.')
```

Scrollbar visibility and offset

`type` controls when the bars appear — `always` keeps them visible, `hover` (the default) shows them only while you interact, `scroll` shows them while you scroll, and `auto` shows them only when content actually overflows. `offsetScrollbars=true` insets the content so the bar never overlaps it; `scrollbarSize` sets the bar thickness in px.

Preview — `type='auto'`, so the bar appears only because this content overflows:

With `offsetScrollbars`, the content is inset so the bar never overlaps it.

This area uses `type='auto'`, so the scrollbar appears only because the content here is taller than 120 pixels — remove a few lines and the bar would disappear.

Source: Markdown

```
{% scrollarea h='120' type='auto' offsetScrollbars=true scrollbarSize=10 %}
With offsetScrollbars, the content is inset so the bar never overlaps it.
{% endScrollarea %}
```

Source: Python

```
component('aardvark', 'scrollarea',
          h='120', type='auto', offsetScrollbars=True, scrollbarSize=10,
          children='With offsetScrollbars, the content is inset so the bar never overlaps
it.')
```

With other components

Wrap a tall stack of Paper surfaces in a `scrollarea` to keep a long list within a fixed-height panel while the rest of the page stays put.

Preview:

First item

Second item

Third item

Fourth item

Fifth item — scroll to see the rest

Source: Markdown

```
{% scrollarea h='140' type='always' p='xs' %}
{% paper withBorder=true p='sm' radius='sm' mb='xs' %}First item{% endPaper %}
{% paper withBorder=true p='sm' radius='sm' %}…more items…{% endPaper %}
{% endScrollarea %}
```

Source: Python

```
items = ''.join(
    component('aardvark', 'paper', withBorder=True, p='sm', radius='sm', mb='xs',
              children=f'Item {i}')
    for i in range(1, 6)
```

```
)
component('aardvark', 'scrollarea', h='140', type='always', p='xs', children=items)
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `offsetScrollbars`) become `=True`.

Attribute	Valid values	Description
`type`	`hover` (default) / `scroll` / `always` / `auto` / `never`	When the scrollbars are shown.
`scrollbarSize`	An integer (px)	Scrollbar thickness.
`scrollHideDelay`	An integer (ms)	Delay before the bars hide (with `type='hover'` / `scroll`).
`offsetScrollbars`	`true` / `false` (default `false`)	Inset the content so the bars don't overlap it.
`h` / `w`	A size token or CSS length	Height / width of the viewport — give it an `h` so the content can overflow.
`miw` / `mih` / `maw` / `mah`	A size token or CSS length	Min/max width and height.
`m` / `mt` / `mb` / `ml` / `mr` / `mx` / `my`	A Mantine size token or CSS length	Margin (all / top / bottom / left / right / horizontal / vertical).
`p` / `pt` / `pb` / `pl` / `pr` / `px` / `py`	A Mantine size token or CSS length	Padding (all / top / bottom / left / right / horizontal / vertical).
`bg`	A theme color or CSS color	Background color.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `scrollarea` carries `data-aardvark-island="ScrollArea"` on its wrapper, and Mantine exposes its parts as `mantine-ScrollArea-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="ScrollArea"] {
  /* style every scrollarea on the page */
```

```
}

.mantine-ScrollArea-root {
  /* the root part */
}

.mantine-ScrollArea-viewport {
  /* the viewport part */
}

.mantine-ScrollArea-scrollbar {
  /* the scrollbar part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Lots of content to scroll through.

Source: Markdown

```
{% scrollarea h='160' type='always' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Lots of content to scroll through.
{% endScrollarea %}
```

Source: Python

```
component('aardvark', 'scrollarea',
          h='160', type='always',
          children='Lots of content to scroll through.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Scroller

`scroller` is a **minimal, low-level scroll container** — a plain overflow box with the **browser's native scrollbars**, no styled overlay bars and no JavaScript. Give it a fixed height with `h` (and/or width with `w`) and its content scrolls; `axis` chooses which way. It's built on Box (it just adds an `overflow` style), so it inherits the same spacing, sizing, and color props. For nicer, styled, auto-hiding overlay scrollbars that look the same across platforms, use ScrollArea — `scroller` is the no-frills sibling. Close the block with `{% endScroller %}` (one capital S).

Use it as `{% scroller %}…{% endScroller %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'scroller', …)`.

Demonstrations

Give it an `h` so the content can overflow; the block body is the content. By default it scrolls vertically (`axis='y'`).

Preview:

A native-scrolling region — the browser's own scrollbar, no styling applied.

This is deliberately plain: it's just a `Box` with `overflow-y: auto`. If you want a styled, auto-hiding scrollbar, switch to the `scrollarea` tag.

Keep reading — there's more content here than fits in 140 pixels, so the native scrollbar appears and you can scroll.

Source: Markdown

```
{% scroller h='140' p='sm' bg='var(--mantine-color-gray-light)' %}
A native-scrolling region — the browser's own scrollbar, no styling applied.
Add enough text and it scrolls vertically inside the fixed 140px height.
{% endScroller %}
```

Source: Python

```
component('aardvark', 'scroller',
          h='140', p='sm', bg='var(--mantine-color-gray-light)',
          children='A native-scrolling region — the browser\'s own scrollbar.')
```

Horizontal scrolling

`axis='x'` scrolls sideways instead — useful for a wide table or a row of items that shouldn't wrap. `axis='both'` enables scrolling on both axes.

Preview:

onetwothreefourfivesixseveneightnineteneleventwelve

Source: Markdown

```
{% scroller axis='x' w='100%' p='sm' bg='var(--mantine-color-gray-light)' %}
<div style="display:flex; gap:1rem;
white-space:nowrap;"><span>one</span>…<span>ten</span></div>
{% endScroller %}
```

Source: Python

```
row = '<div style="display:flex; gap:1rem; white-space:nowrap;">' + \
      ''.join(f'<span>{n}</span>' for n in ['one', 'two', 'three', 'four', 'five']) + \
      '</div>'
component('aardvark', 'scroller', axis='x', w='100%', p='sm', children=row)
```

With other components

Wrap a stack of Paper surfaces in a fixed-height `scroller` to keep a long list inside a native-scrolling panel.

Preview:

First row

Second row

Third row

Fourth row

Fifth row — scroll for more

Source: Markdown

```
{% scroller h='140' p='xs' bg='var(--mantine-color-gray-light)' %}
{% paper withBorder=true p='sm' radius='sm' mb='xs' %}First row{% endPaper %}
{% paper withBorder=true p='sm' radius='sm' %}…more rows…{% endPaper %}
{% endScroller %}
```

Source: Python

```
rows = ''.join(
    component('aardvark', 'paper', withBorder=True, p='sm', radius='sm', mb='xs',
              children=f'Row {i}')
    for i in range(1, 6)
)
component('aardvark', 'scroller', h='140', p='xs', children=rows)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`axis`	`y` (default, vertical) / `x` (horizontal) / `both`	Which way the box scrolls.
`h` / `w`	A CSS length, or a bare number (treated as pixels)	Height / width — give it an `h` so the content can overflow.
`miw` / `mih` / `maw` / `mah`	A CSS length, or a bare number (treated as pixels)	Min/max width and height.
`bg`	A theme color or CSS color	Background color.
`m` / `mt` / `mb` / `ml` / `mr` / `mx` / `my`	A Mantine size token or CSS length	Margin (all / top / bottom / left / right / horizontal / vertical).
`p` / `pt` / `pb` / `pl` / `pr` / `px` / `py`	A Mantine size token or CSS length	Padding (all / top / bottom / left / right / horizontal / vertical).
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selectors

Each `scroller` carries `data-aardvark-island="Box"` on its wrapper; it renders a single element with no Mantine Styles API parts, so target the island wrapper.

```
[data-aardvark-island="Box"] {
  /* style every scroller on the page */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Lots of content to scroll through.

Source: Markdown

```
{% scroller h='140' p='sm' bg='var(--mantine-color-gray-light)' attr={'onclick': '''
```

```
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Lots of content to scroll through.
{% endScroller %}
```

Source: Python

```
component('aardvark', 'scroller',
          h='140', p='sm', bg='var(--mantine-color-gray-light)',
          children='Lots of content to scroll through.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

SimpleGrid

`{% simplegrid %}` lays its children out in a grid where **every column is the same width** — the simplest way to arrange a row of equal cards or tiles that reflows to fewer columns on narrow screens. Set the base column count with `cols`, the gaps with `spacing` (horizontal) and `verticalSpacing` (vertical), and let any of those adapt per breakpoint with the `*Sm`/`*Md`/`*Lg`/`*Xl` props. Each top-level block in the body becomes one grid cell.

Use it as `{% simplegrid %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'simplegrid', …)`.

Fixed columns

`cols` sets the number of equal-width columns. Here three cells sit in three columns with the default `md` spacing.

[One] [Two] [Three]

Source: Markdown

```
{% simplegrid cols=3 spacing='md' %}
{% badge size='lg' fullWidth %}One{% endBadge %}

{% badge size='lg' fullWidth %}Two{% endBadge %}

{% badge size='lg' fullWidth %}Three{% endBadge %}
{% endSimplegrid %}
```

Source: Python

```
component('aardvark', 'simplegrid', cols=3, spacing='md', children=(
    component('aardvark', 'badge', size='lg', fullWidth=True, children='One')
    + '\n\n'
    + component('aardvark', 'badge', size='lg', fullWidth=True, children='Two')
    + '\n\n'
    + component('aardvark', 'badge', size='lg', fullWidth=True, children='Three')
))
```

Responsive columns

`cols` is the base column count; `colsSm` / `colsMd` / `colsLg` / `colsXl` override it at that Mantine breakpoint and up. The grid below is one column on a phone, two on a tablet, and four on a wide screen — resize the window to watch it reflow.

[A] [B] [C] [D]

Source: Markdown

```
{% simplegrid cols=1 colsSm=2 colsLg=4 spacing='sm' %}
{% badge fullWidth %}A{% endBadge %}

{% badge fullWidth %}B{% endBadge %}

{% badge fullWidth %}C{% endBadge %}

{% badge fullWidth %}D{% endBadge %}
{% endSimplegrid %}
```

Source: Python

```
component('aardvark', 'simplegrid', cols=1, colsSm=2, colsLg=4, spacing='sm', children=(
    component('aardvark', 'badge', fullWidth=True, children='A')
    + '\n\n'
    + component('aardvark', 'badge', fullWidth=True, children='B')
    + '\n\n'
    + component('aardvark', 'badge', fullWidth=True, children='C')
    + '\n\n'
    + component('aardvark', 'badge', fullWidth=True, children='D')
))
```

Spacing and vertical spacing

`spacing` is the horizontal gap between columns; `verticalSpacing` is the gap between rows (it defaults to `spacing` when omitted). Each is a Mantine size token or any CSS value, and each takes per-breakpoint overrides — `spacingSm`, `verticalSpacingLg`, and so on.

[One] [Two] [Three] [Four]

Source: Markdown

```
{% simplegrid cols=2 spacing='xl' verticalSpacing='xs' %}
{% badge fullWidth %}One{% endBadge %}

{% badge fullWidth %}Two{% endBadge %}

{% badge fullWidth %}Three{% endBadge %}

{% badge fullWidth %}Four{% endBadge %}
{% endSimplegrid %}
```

Source: Python

```
component('aardvark', 'simplegrid', cols=2, spacing='xl', verticalSpacing='xs', children=(
    component('aardvark', 'badge', fullWidth=True, children='One')
    + '\n\n'
    + component('aardvark', 'badge', fullWidth=True, children='Two')
```

```
    + '\n\n'
    + component('aardvark', 'badge', fullWidth=True, children='Three')
    + '\n\n'
    + component('aardvark', 'badge', fullWidth=True, children='Four')
))
```

With other components

Each cell can hold any component, not just a badge. Here the grid arranges three `card`s that reflow from three columns to one on a phone.

Fast

Builds in seconds.

Typed

Schema-checked config.

Themed

One palette, everywhere.

Source: Markdown

```
{% simplegrid cols=3 colsSm=1 spacing='md' %}
{% card title='Fast' %}Builds in seconds.{% endCard %}

{% card title='Typed' %}Schema-checked config.{% endCard %}

{% card title='Themed' %}One palette, everywhere.{% endCard %}
{% endSimplegrid %}
```

Source: Python

```
component('aardvark', 'simplegrid', cols=3, colsSm=1, spacing='md', children=(
    component('aardvark', 'card', title='Fast', children='Builds in seconds.')
    + '\n\n'
    + component('aardvark', 'card', title='Typed', children='Schema-checked config.')
    + '\n\n'
    + component('aardvark', 'card', title='Themed', children='One palette, everywhere.')
))
```

Attributes

Attribute	Valid values	Description
`cols`	integer (`1` default)	Base number of equal-width columns.
`colsSm`, `colsMd`, `colsLg`, `colsXl`	integer	Column count at that breakpoint and up.
`spacing`	`xs`, `sm`, `md` (default), `lg`, `xl`, or any CSS value	Horizontal gap between columns.
`spacingSm`, `spacingMd`, `spacingLg`, `spacingXl`	Mantine size token or any CSS value	Horizontal gap at that breakpoint and up.
`verticalSpacing`	Mantine size token or any CSS value	Vertical gap between rows (defaults to `spacing`).
`verticalSpacingSm`, `verticalSpacingMd`, `verticalSpacingLg`, `verticalSpacingXl`	Mantine size token or any CSS value	Vertical gap at that breakpoint and up.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	Mantine size token or any CSS value	Margin (all sides / top / bottom / left / right / horizontal / vertical).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	Mantine size token or any CSS value	Padding (all sides / top / bottom / left / right / horizontal / vertical).
`bg`	color name or CSS color	Background color.
`c`	color name or CSS color	Text color.
`w`, `h`	Mantine size token or any CSS value	Width / height.

`miw`, `mih`, `maw`, `mah`	any CSS value	Min/max width and min/max height.

CSS Selectors

Each `simplegrid` carries `data-aardvark-island="SimpleGrid"` on its wrapper, and Mantine exposes its parts as `mantine-SimpleGrid-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="SimpleGrid"] {
  /* style every simplegrid on the page */
}

.mantine-SimpleGrid-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

A

B

C

Source: Markdown

```
{% simplegrid cols=3 spacing='md' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% paper %}A{% endPaper %} {% paper %}B{% endPaper %} {% paper %}C{% endPaper %}
{% endSimplegrid %}
```

Source: Python

```
component('aardvark', 'simplegrid', cols=3, spacing='md', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}, children=(
    component('aardvark', 'paper', children='A')
    + ' '
    + component('aardvark', 'paper', children='B')
    + ' '
```

```
    + component('aardvark', 'paper', children='C')
))
```

Space

`{% space %}` is an **empty spacer** — it adds horizontal or vertical room between elements without reaching for a margin. Set `h` for vertical space (above and below) or `w` for horizontal space (between inline elements); each takes a Mantine size token (`xs`–`xl`) or any CSS value. It is self-closing — there is no body.

Use it as `{% space %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'space', …)`.

Vertical space

`h` adds vertical room between stacked elements.

[Above] [Below]

Source: Markdown

```
{% badge %}Above{% endBadge %}
{% space h='xl' %}
{% badge %}Below{% endBadge %}
```

Source: Python

```
component('aardvark', 'badge', children='Above')
component('aardvark', 'space', h='xl')
component('aardvark', 'badge', children='Below')
```

Horizontal space

`w` adds room between inline elements. Wrapping the row in a `group` with `gap=0` lets the spacer do all the spacing.

[Left][Right]

Source: Markdown

```
{% group gap=0 %}
{% badge %}Left{% endBadge %}{% space w='xl' %}{% badge %}Right{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', gap='0', children=(
    component('aardvark', 'badge', children='Left')
    + component('aardvark', 'space', w='xl')
    + component('aardvark', 'badge', children='Right')
))
```

CSS value

Both `h` and `w` accept any CSS length, not just the `xs`–`xl` tokens — handy when you need an exact gap.

[48px above the next badge] [Here]

Source: Markdown

```
{% badge %}48px above the next badge{% endBadge %}
{% space h='48px' %}
{% badge %}Here{% endBadge %}
```

Source: Python

```
component('aardvark', 'badge', children='48px above the next badge')
component('aardvark', 'space', h='48px')
component('aardvark', 'badge', children='Here')
```

With other components

`space` sits between any two components. Here it adds breathing room between a `button` and the `badge` that follows it.

Run

[Idle]

Source: Markdown

```
{% group gap=0 align='center' %}
{% button text='Run' %}{% space w='lg' %}{% badge color='green' %}Idle{% endBadge %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'group', gap='0', align='center', children=(
    component('aardvark', 'button', text='Run')
    + component('aardvark', 'space', w='lg')
    + component('aardvark', 'badge', color='green', children='Idle')
))
```

For finer control, the Mantine spacing props on any other tag (`mt`, `mb`, `mx`, ...) cover most cases without a separate spacer.

Attributes

Attribute	Valid values	Description
`h`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	Vertical space (height of the spacer).
`w`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	Horizontal space (width of the spacer).

CSS Selectors

Each `space` carries `data-aardvark-island="Space"` on its wrapper; it renders a single spacer element with no Mantine Styles API parts, so target the island wrapper.

```
[data-aardvark-island="Space"] {
  /* style every space on the page */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Source: Markdown

```
{% space h='xl' attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'space', h='xl', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

Splitter

`{% splitter %}` places **two panels side by side** (or stacked) with a **draggable divider** between them — a quick way to show two resizable regions of content together, like a label column beside a detail column. Provide the panels as the `first` and `second` text params, or put the first panel in the block body and the second in `second`. Set the first panel's initial size with `defaultSize` and lay the panels out with `orientation`; drag the handle (or use the arrow keys when it's focused) to resize.

Use it as `{% splitter %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'splitter', …)`.

Body as the first panel

The block body becomes the first panel; `second` holds the second. `defaultSize` is the first panel's initial size as a percentage (`50` default); drag the handle to change it.

The first panel is 40% wide.

Source: Markdown

```
{% splitter second='The second panel takes the remaining space.' defaultSize=40 %}
The first panel is 40% wide.
{% endSplitter %}
```

Source: Python

```
component(
    'aardvark', 'splitter',
    second='The second panel takes the remaining space.',
    defaultSize=40,
    children='The first panel is 40% wide.',
)
```

Both panels as params

Set `first` and `second` to skip the block body entirely — convenient when both panels are short strings or when you are calling from Python.

Source: Markdown

```
{% splitter first='Label column' second='Detail column with the bulk of the content.'
defaultSize=30 %}
```

Source: Python

```
component(
    'aardvark', 'splitter',
    first='Label column',
    second='Detail column with the bulk of the content.',
    defaultSize=30,
)
```

Stacked (vertical)

`orientation='vertical'` stacks the panels with a horizontal divider between them; `defaultSize` then sets the first (top) panel's initial height.

Source: Markdown

```
{% splitter first='Top panel' second='Bottom panel' orientation='vertical' defaultSize=50 %}
```

Source: Python

```
component(
    'aardvark', 'splitter',
    first='Top panel',
    second='Bottom panel',
    orientation='vertical',
    defaultSize=50,
)
```

With other components

The block-body panel renders as Markdown, so the first panel can hold any other component. Here it carries a `badge` heading above a short description.

[Overview]

A short summary lives in the first panel.

Source: Markdown

```
{% splitter second='The detail panel sits to the right and fills the rest of the row.'
defaultSize=35 %}
{% badge color='grape' %}Overview{% endBadge %}

A short summary lives in the first panel.
{% endSplitter %}
```

Source: Python

```
component(
    'aardvark', 'splitter',
```

```
    second='The detail panel sits to the right and fills the rest of the row.',
    defaultSize=35,
    children=(
        component('aardvark', 'badge', color='grape', children='Overview')
        + '\n\nA short summary lives in the first panel.'
    ),
)
```

Attributes

Attribute	Valid values	Description
`first`	text	First panel content. Omit it to use the block body as the first panel instead.
`second`	text	Second panel content.
`defaultSize`	percentage `0`–`100` (`50` default)	The first panel's initial width (`horizontal`) or height (`vertical`); drag the handle to change it.
`orientation`	`horizontal` (default), `vertical`	`horizontal` places panels side by side; `vertical` stacks them.

CSS Selectors

Each `splitter` carries `data-aardvark-island="Splitter"` on its wrapper, and Mantine exposes its parts as `mantine-Splitter-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Splitter"] {
  /* style every splitter on the page */
}

.mantine-Splitter-root {
  /* the root part */
}

.mantine-Splitter-pane {
  /* the pane part */
}

.mantine-Splitter-handle {
  /* the handle part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

The first panel.

Source: Markdown

```
{% splitter second='The second panel takes the remaining space.' defaultSize=40
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
The first panel.
{% endSplitter %}
```

Source: Python

```
component(
    'aardvark', 'splitter',
    second='The second panel takes the remaining space.',
    defaultSize=40,
    attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
    children='The first panel.',
)
```

Stack

`{% stack %}` stacks its children in a **vertical column** with a consistent gap — the column counterpart to `{% group %}`. It is a flex column: control where children sit along the column (`justify`), how they line up across it (`align`), and the space between them (`gap`). By default `align` is `stretch`, so children fill the width. The block body is the content.

Use it as `{% stack %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'stack', …)`.

Basic column

With no options, children stretch to the full width and stack with the default `md` gap.

Top

Middle

Bottom

Source: Markdown

```
{% stack %}
{% button text='Top' %}
{% button text='Middle' %}
{% button text='Bottom' %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'stack', children=(
    component('aardvark', 'button', text='Top')
    + component('aardvark', 'button', text='Middle')
    + component('aardvark', 'button', text='Bottom')
))
```

Align (cross axis)

`align` sets `align-items` — how children line up across the column. The default is `stretch` (children fill the width); use `center`, `flex-start`, or `flex-end` to size children to their content and position them.

[One] [Two] [Three]

Source: Markdown

```
{% stack align='center' %}
```

```
{% badge %}One{% endBadge %}
{% badge %}Two{% endBadge %}
{% badge %}Three{% endBadge %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'stack', align='center', children=(
    component('aardvark', 'badge', children='One')
    + component('aardvark', 'badge', children='Two')
    + component('aardvark', 'badge', children='Three')
))
```

Justify (main axis)

`justify` sets `justify-content` — where children sit along the column when the stack is taller than its content. The default is `flex-start`; `center`, `flex-end`, and `space-between` need a fixed height (`h`) to have a visible effect.

[First] [Last]

Source: Markdown

```
{% stack justify='space-between' h=160 align='center' %}
{% badge %}First{% endBadge %}
{% badge %}Last{% endBadge %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'stack', justify='space-between', h=160, align='center', children=(
    component('aardvark', 'badge', children='First')
    + component('aardvark', 'badge', children='Last')
))
```

Gap

`gap` is the space between children — a Mantine size token (`xs`–`xl`) or any CSS value (`4px`, `2rem`). The default is `md`.

[One] [Two] [Three]

Source: Markdown

```
{% stack gap='xs' align='center' %}
{% badge %}One{% endBadge %}
{% badge %}Two{% endBadge %}
```

```
{% badge %}Three{% endBadge %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'stack', gap='xs', align='center', children=(
    component('aardvark', 'badge', children='One')
    + component('aardvark', 'badge', children='Two')
    + component('aardvark', 'badge', children='Three')
))
```

With other components

`stack` composes with any other tag. Here it stacks a heading badge above a `group` of action `button`s.

[Project settings]

Save

Cancel

Source: Markdown

```
{% stack gap='sm' %}
{% badge size='lg' color='grape' %}Project settings{% endBadge %}
{% group %}
{% button text='Save' %} {% button text='Cancel' variant='default' %}
{% endGroup %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'stack', gap='sm', children=(
    component('aardvark', 'badge', size='lg', color='grape', children='Project settings')
    + component('aardvark', 'group', children=(
        component('aardvark', 'button', text='Save')
        + component('aardvark', 'button', text='Cancel', variant='default')
    ))
))
```

Attributes

Attribute	Valid values	Description
`justify`	`flex-start` (default), `center`, `flex-end`, `space-between`, `space-around`, `space-evenly`	`justify-content` for the column — where children sit on the main axis (needs a fixed height to show).
`align`	`stretch` (default), `center`, `flex-start`, `flex-end`	`align-items` — how children line up on the cross axis.
`gap`	`xs`, `sm`, `md` (default), `lg`, `xl`, or any CSS value	Space between children.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	Mantine size token or any CSS value	Margin (all sides / top / bottom / left / right / horizontal / vertical).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	Mantine size token or any CSS value	Padding (all sides / top / bottom / left / right / horizontal / vertical).
`bg`	color name or CSS color	Background color.
`c`	color name or CSS color	Text color.
`w`, `h`	Mantine size token or any CSS value	Width / height.
`miw`, `mih`, `maw`, `mah`	any CSS value	Min/max width and min/max height.

CSS Selectors

Each `stack` carries `data-aardvark-island="Stack"` on its wrapper, and Mantine exposes its parts as `mantine-Stack-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Stack"] {
  /* style every stack on the page */
}

.mantine-Stack-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

[one] [two]

Source: Markdown

```
{% stack attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% badge %}one{% endBadge %} {% badge %}two{% endBadge %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'stack', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}, children=(
    component('aardvark', 'badge', children='one')
    + ' '
    + component('aardvark', 'badge', children='two')
))
```

Inputs

Everything for collecting input, each a single **built-in** tag — no setup, no JavaScript to write. The text-entry **fields** wrap the matching Mantine inputs and expose their full surface (label, description, error, size, radius, variant, required, disabled, plus each field's own props); the **controls** mount as interactive islands you can toggle, drag, and pick right on the page.

Fields

- TextInput — a single-line text field.
- Textarea — a multi-line text field with autosize.
- NumberInput — numeric entry with min/max/step, prefix/suffix, and controls.
- PasswordInput — a password field with a show/hide toggle.
- JsonInput — a textarea that validates and reformats JSON.
- MaskInput — formatted entry against a fixed pattern (phone, date, card).
- FileInput — a file picker with accept, multiple, and clearable.
- Dropzone — a drag-and-drop file upload zone with a built-in drag-state overlay.
- PinInput — a row of single-character boxes for codes / PINs.
- NativeSelect — a native `<select>` from a delimited option list.
- Fieldset — a bordered group of fields with a legend.
- Input — the unstyled base input primitive every field is built on.
- Form — the `useForm` hook tying fields together: validation, submit, and collected values.

Toggles and choices

- Checkbox — a single labeled checkbox.
- Radio — a labeled radio button; share a `name` for a group.
- Switch — a toggle switch with optional on/off track text.
- Chip — a checkbox or radio styled as a pill.
- SegmentedControl — a row of mutually exclusive options.
- Rating — a star rating for feedback or scores.

Rich text

- RichTextEditor — a live in-page editor with a formatting toolbar (bold, italic, underline, lists, links), backed by Tiptap.

Sliders

- Slider — pick a number in a range, with marks.
- RangeSlider — pick a range with two thumbs.
- AngleSlider — pick an angle on a dial (0–360°).
- HueSlider — pick a hue along the spectrum.
- AlphaSlider — pick an alpha (opacity) for a color.

Color pickers

- ColorInput — a text input with a swatch and dropdown picker.
- ColorPicker — an inline saturation + hue (+ alpha) picker.

Focus

- FocusTrap — keep keyboard focus inside a region (the focus behavior behind modals).

AlphaSlider

A **built-in** tag for an alpha slider — a thin track that fades a color from transparent to opaque, for picking an alpha (opacity) from 0 to 1. It's one of Mantine's color-picker building blocks. It ships with Aardvark, so it's a single tag with no setup. Set the base `color` so the track can show the fade, and `defaultValue` for the starting opacity.

Use it as `{% alphaslider %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'alphaslider', …)`.

Demonstrations

Basic alpha

Set the base `color` (so the track can render its fade) and a `defaultValue` from 0 (transparent) to 1 (opaque). Drag to change the opacity.

Source: Markdown

```
{% alphaslider color='#1c7ed6' defaultValue=0.6 %}
```

Source: Python

```
component('aardvark', 'alphaslider', color='#1c7ed6', defaultValue=0.6)
```

Colors, sizes, and radius

Any base `color` paints the track's fade; `size` sets the track thickness (`xs`–`xl`) and `radius` rounds its ends (`xs`–`xl`).

Source: Markdown

```
{% alphaslider color='#fa5252' defaultValue=0.4 size='xl' radius='xl' %}
{% alphaslider color='#40c057' defaultValue=0.8 size='sm' %}
```

Source: Python

```
component('aardvark', 'alphaslider', color='#fa5252', defaultValue=0.4, size='xl',
radius='xl')
component('aardvark', 'alphaslider', color='#40c057', defaultValue=0.8, size='sm')
```

With other components

Generate one alpha slider per swatch from data with a Python loop — the same `component('aardvark', 'alphaslider', …)` call, once per item, laid out in a card grid.

Primary

Success

Danger

Source: Python

```
swatches = [
    {"name": "Primary", "color": "#1c7ed6", "alpha": 0.9},
    {"name": "Success", "color": "#40c057", "alpha": 0.6},
    {"name": "Danger", "color": "#fa5252", "alpha": 0.4},
]
controls = ''
for s in swatches:
    row = '**' + s["name"] + '**\n\n' + component(
        'aardvark', 'alphaslider', color=s["color"], defaultValue=s["alpha"], size='lg',
    )
    controls += component('aardvark', 'card', variant='plain', children=row)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=controls))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description
`color`	color string	The base color the alpha applies to (e.g. `#1c7ed6`). Required for the fade track to render.
`defaultValue`	number `0`–`1`	Starting alpha — the reader can drag it. `0` is transparent, `1` is opaque.
`size`	`xs`–`xl`	Track thickness.
`radius`	`xs`–`xl`	Corner rounding of the track ends.
`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

CSS Selectors

Target a `{% alphaslider %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every AlphaSlider instance on the page */
[data-aardvark-island="AlphaSlider"] { }

/* Mantine Styles API parts */
.mantine-AlphaSlider-slider { }
.mantine-AlphaSlider-sliderOverlay { }
.mantine-AlphaSlider-thumb { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so dragging the alpha control reads the changed control value, logs it to the console, and alerts it:

Source: Markdown

```
{% alphaslider color='#1c7ed6' defaultValue=0.6 attr={'onchange': '''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'alphaslider', color='#1c7ed6', defaultValue=0.6, attr={'onchange':
'''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''})
```

AngleSlider

A **built-in** tag for an angle slider — a circular dial for picking an angle from 0 to 360 degrees. It ships with Aardvark, so it's a single tag with no setup. Set the dial diameter with `size`, the starting angle with `defaultValue`, and the increment with `step`.

Use it as `{% angleslider %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'angleslider', …)`.

Demonstrations

Basic dial

Set `defaultValue` (in degrees) and a `size`. Drag around the dial to change it.

Source: Markdown

```
{% angleslider size=80 defaultValue=90 %}
```

Source: Python

```
component('aardvark', 'angleslider', size=80, defaultValue=90)
```

Stepping and marks

`step` snaps the angle to a fixed increment; `marks` adds labelled ticks as a JSON array of `{value, label}`. A larger `thumbSize` makes the handle easier to grab.

Source: Markdown

```
{% angleslider size=100 step=45 thumbSize=12 defaultValue=45 marks='[{"value": 0, "label":
"N"}, {"value": 90, "label": "E"}, {"value": 180, "label": "S"}, {"value": 270, "label":
"W"}]' %}
```

Source: Python

```
component('aardvark', 'angleslider',
          size=100, step=45, thumbSize=12, defaultValue=45,
          marks='[{"value": 0, "label": "N"}, {"value": 90, "label": "E"}, {"value": 180,
"label": "S"}, {"value": 270, "label": "W"}]')
```

No center label, and disabled

The center shows the current degree label by default. Set `withLabel=false` to hide it, and add the bare `disabled` flag for a read-only dial.

Source: Markdown

```
{% angleslider size=80 defaultValue=135 withLabel=false %}
{% angleslider size=80 defaultValue=200 disabled=true %}
```

Source: Python

```
component('aardvark', 'angleslider', size=80, defaultValue=135, withLabel=False)
component('aardvark', 'angleslider', size=80, defaultValue=200, disabled=True)
```

With other components

Generate one dial per axis from data with a Python loop — the same `component('aardvark', 'angleslider', …)` call, once per item, laid out in a card grid.

Heading

Pitch

Roll

Source: Python

```
axes = [
    {"name": "Heading", "value": 45},
    {"name": "Pitch", "value": 270},
    {"name": "Roll", "value": 135},
]
dials = ''
for a in axes:
    row = '**' + a["name"] + '**\n\n' + component(
        'aardvark', 'angleslider', size=90, step=15, defaultValue=a["value"],
    )
    dials += component('aardvark', 'card', variant='plain', children=row)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=dials))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description

`size`	integer	Diameter of the dial in pixels.
`step`	integer	Increment per move, in degrees.
`defaultValue`	integer	Starting angle in degrees — the reader can drag it.
`thumbSize`	integer	Diameter of the thumb in pixels.
`marks`	JSON array string	Labelled ticks as `[{"value": n, "label": "…"}, …]`.
`withLabel`	boolean	Show the degree label in the center (on by default; set `false` to hide).
`disabled`	boolean	Make the dial read-only.
`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

The center label format (`formatLabel` in Mantine) is a JavaScript function rather than a value, so it can't be set from a build-time tag or `component('aardvark', 'angleslider', …)` call — the dial shows the raw degree value. Toggle it off entirely with `withLabel=false`.

CSS Selectors

Target a `{% angleslider %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every AngleSlider instance on the page */
[data-aardvark-island="AngleSlider"] { }

/* Mantine Styles API parts */
.mantine-AngleSlider-root { }
.mantine-AngleSlider-thumb { }
.mantine-AngleSlider-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so turning the dial reads the changed control value, logs it to the console, and alerts it:

Source: Markdown

```
{% angleslider size=80 defaultValue=90 attr={'onchange': '''
```

```
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'angleslider', size=80, defaultValue=90, attr={'onchange': '''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''})
```

Checkbox

A single checkbox with a label. The label is the `label` attribute or the block body, and the box is interactive — readers can toggle it. Add `defaultChecked` to start it ticked, or `indeterminate` for the mixed state.

Use it as `{% checkbox %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'checkbox', …)`.

Label and default state

The label comes from the `label` attribute or the block body. Add `defaultChecked` to start the box ticked.

Accept the terms

Subscribe to updates

Source: Markdown

```
{% checkbox label='Accept the terms' %}

{% checkbox defaultChecked=true color='green' %}Subscribe to updates{% endCheckbox %}
```

Source: Python

```
component('aardvark', 'checkbox', label='Accept the terms')

component('aardvark', 'checkbox', defaultChecked=True, color='green',
          children='Subscribe to updates')
```

Colors and sizes

`color` is any theme color for the checked fill; `size` is `xs`–`xl`.

blue

grape

large

Source: Markdown

```
{% checkbox label='blue' defaultChecked=true %}
{% checkbox label='grape' color='grape' defaultChecked=true %}
{% checkbox label='large' size='lg' defaultChecked=true %}
```

Source: Python

```
component('aardvark', 'checkbox', label='blue', defaultChecked=True)
component('aardvark', 'checkbox', label='grape', color='grape', defaultChecked=True)
component('aardvark', 'checkbox', label='large', size='lg', defaultChecked=True)
```

Label position, indeterminate, and disabled

`labelPosition='left'` puts the label before the box; `indeterminate` shows the mixed/partial state; `disabled` makes it non-interactive.

Label on the left

Indeterminate

Disabled

Source: Markdown

```
{% checkbox label='Label on the left' labelPosition='left' defaultChecked=true %}
{% checkbox label='Indeterminate' indeterminate=true %}
{% checkbox label='Disabled' disabled=true defaultChecked=true %}
```

Source: Python

```
component('aardvark', 'checkbox', label='Label on the left',
          labelPosition='left', defaultChecked=True)
component('aardvark', 'checkbox', label='Indeterminate', indeterminate=True)
component('aardvark', 'checkbox', label='Disabled', disabled=True, defaultChecked=True)
```

Radius, icon color, and helper text

`radius` rounds the corners; `iconColor` colors the check mark itself; `description` and `error` add helper text below the label.

Rounded

Custom check color

With a description

With an error

Source: Markdown

```
{% checkbox label='Rounded' radius='xl' defaultChecked=true %}
{% checkbox label='Custom check color' iconColor='yellow' color='dark' defaultChecked=true
%}
{% checkbox label='With a description' description='We only email about outages.'
defaultChecked=true %}
{% checkbox label='With an error' error='You must accept to continue.' %}
```

Source: Python

```
component('aardvark', 'checkbox', label='Rounded', radius='xl', defaultChecked=True)
component('aardvark', 'checkbox', label='Custom check color', iconColor='yellow',
          color='dark', defaultChecked=True)
component('aardvark', 'checkbox', label='With a description',
          description='We only email about outages.', defaultChecked=True)
component('aardvark', 'checkbox', label='With an error',
          error='You must accept to continue.')
```

With other components

Pair a checkbox with surrounding text and a button to build a small consent row.

Please review and confirm before continuing.

I have read the privacy policy

Continue

Source: Markdown

```
Please review and confirm before continuing.

{% checkbox label='I have read the privacy policy' defaultChecked=true %}

{% button %}Continue{% endButton %}
```

Source: Python

```
component('aardvark', 'checkbox', label='I have read the privacy policy',
          defaultChecked=True)
component('aardvark', 'button', children='Continue')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	The label, when not using the block body. Plain text — Markdown (e.g. `**bold**`) is not rendered.
`defaultChecked`	bare flag (`true`)	Start the box ticked. It stays interactive — the reader can toggle it.

`indeterminate`	bare flag (`true`)	Show the mixed/partial state (a dash instead of a check).
`disabled`	bare flag (`true`)	Render non-interactive.
`color`	theme color name (`blue`, `green`, `grape`, …)	Fill color when checked.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Overall size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner rounding of the box.
`labelPosition`	`right` (default), `left`	Which side of the box the label sits on.
`iconColor`	theme color name	Color of the check mark itself.
`description`	string	Helper text shown below the label.
`error`	string	Error text shown below the label (also marks the box invalid).
`autoContrast`	bare flag (`true`)	Auto-pick a readable check-mark color against `color`.
`attr`	dict (`attr={…}`)	Raw HTML attributes (e.g. `onchange`) applied to the rendered root.

CSS Selectors

Target a `{% checkbox %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Checkbox instance on the page */
[data-aardvark-island="Checkbox"] { }

/* Mantine Styles API parts */
.mantine-Checkbox-root { }
.mantine-Checkbox-input { }
.mantine-Checkbox-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so toggling it logs its checked state to the console and alerts it:

Accept the terms

Source: Markdown

```
{% checkbox label='Accept the terms' attr={'onchange': '''
const value = this.checked;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'checkbox', label='Accept the terms', attr={'onchange': '''
const value = this.checked;
console.log('attr demo value:', value);
alert(value);
'''})
```

Chip

A chip — a checkbox or radio styled as a selectable pill. The label is the block body or the `text` attribute, and the chip is interactive — readers can toggle it. Add `defaultChecked` to start it selected, and pick `type` (`checkbox` or `radio`).

Use it as `{% chip %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'chip', …)`.

Label and default state

The label is the block body or the `text` attribute. `defaultChecked` starts it selected.

React

Selected

Vue

Source: Markdown

```
{% chip %}React{% endChip %}
{% chip defaultChecked=true color='grape' %}Selected{% endChip %}
{% chip text='Vue' %}
```

Source: Python

```
component('aardvark', 'chip', children='React')
component('aardvark', 'chip', defaultChecked=True, color='grape', children='Selected')
component('aardvark', 'chip', text='Vue')
```

In a loop, render a row of chips from data:

Source: Python

```
for tag in ['React', 'Vue', 'Svelte']:
    component('aardvark', 'chip', children=tag)
```

Variants

`variant` is `filled` (default), `light`, or `outline`.

filled

light

outline

Source: Markdown

```
{% chip defaultChecked=true %}filled{% endChip %}
{% chip variant='light' defaultChecked=true %}light{% endChip %}
{% chip variant='outline' defaultChecked=true %}outline{% endChip %}
```

Source: Python

```
component('aardvark', 'chip', defaultChecked=True, children='filled')
component('aardvark', 'chip', variant='light', defaultChecked=True, children='light')
component('aardvark', 'chip', variant='outline', defaultChecked=True, children='outline')
```

Colors, sizes, and radius

`color` is any theme color; `size` is `xs`–`xl`; `radius` rounds the pill.

green

orange

large

less round

Source: Markdown

```
{% chip color='green' defaultChecked=true %}green{% endChip %}
{% chip color='orange' defaultChecked=true %}orange{% endChip %}
{% chip size='lg' defaultChecked=true %}large{% endChip %}
{% chip radius='sm' defaultChecked=true %}less round{% endChip %}
```

Source: Python

```
component('aardvark', 'chip', color='green', defaultChecked=True, children='green')
component('aardvark', 'chip', color='orange', defaultChecked=True, children='orange')
component('aardvark', 'chip', size='lg', defaultChecked=True, children='large')
component('aardvark', 'chip', radius='sm', defaultChecked=True, children='less round')
```

Type and disabled

`type='radio'` gives the chip a round selection marker (use it for one-of-many sets); `disabled` makes it non-interactive.

radio chip

disabled

disabled, selected

Source: Markdown

```
{% chip type='radio' defaultChecked=true %}radio chip{% endChip %}
{% chip disabled=true %}disabled{% endChip %}
{% chip disabled=true defaultChecked=true %}disabled, selected{% endChip %}
```

Source: Python

```
component('aardvark', 'chip', type='radio', defaultChecked=True, children='radio chip')
component('aardvark', 'chip', disabled=True, children='disabled')
component('aardvark', 'chip', disabled=True, defaultChecked=True,
          children='disabled, selected')
```

With other components

Combine chips with a heading and a button to build a small filter row.

Filter by framework

React

Vue

Svelte

Apply

Source: Markdown

```
Filter by framework

{% chip defaultChecked=true %}React{% endChip %}
{% chip %}Vue{% endChip %}
{% chip %}Svelte{% endChip %}

{% button %}Apply{% endButton %}
```

Source: Python

```
component('aardvark', 'chip', defaultChecked=True, children='React')
component('aardvark', 'chip', children='Vue')
component('aardvark', 'chip', children='Svelte')
component('aardvark', 'button', children='Apply')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`text`	string	The label, when not using the block body.
`variant`	`filled` (default), `light`, `outline`	Visual style of the pill.
`color`	theme color name (`blue`, `green`, ...)	Fill / accent color when selected.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Overall size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner rounding of the pill.
`type`	`checkbox` (default), `radio`	Selection marker shape and grouping semantics.
`defaultChecked`	bare flag (`true`)	Start the chip selected. It stays interactive — the reader can toggle it.
`disabled`	bare flag (`true`)	Render non-interactive.
`autoContrast`	bare flag (`true`)	Auto-pick a readable label color against `color`.
`attr`	dict (`attr={…}`)	Raw HTML attributes (e.g. `onchange`) applied to the rendered root.

CSS Selectors

Target a `{% chip %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Chip instance on the page */
[data-aardvark-island="Chip"] { }

/* Mantine Styles API parts */
.mantine-Chip-root { }
.mantine-Chip-label { }
.mantine-Chip-input { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so toggling it logs its checked state to the console and alerts it:

React

Source: Markdown

```
{% chip attr={'onchange': '''
const value = this.checked;
console.log('attr demo value:', value);
alert(value);
'''} %}React{% endChip %}
```

Source: Python

```
component('aardvark', 'chip', children='React', attr={'onchange': '''
const value = this.checked;
console.log('attr demo value:', value);
alert(value);
'''})
```

ColorInput

A **built-in** tag for a color input — a text field with a color swatch and a dropdown color picker. It ships with Aardvark, so it's a single tag with no setup. Give it a `label`, a `format` (how the value reads back), and an optional `defaultValue`; the reader can type a value, pick from the dropdown, or use the eye-dropper.

Use it as `{% colorinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'colorinput', …)`.

Demonstrations

Basic input

Add a `label`, a `format`, and an optional `defaultValue`. Click the swatch to open the picker.

Brand color

Source: Markdown

```
{% colorinput label='Brand color' format='hex' defaultValue='#1c7ed6' %}
```

Source: Python

```
component('aardvark', 'colorinput', label='Brand color', format='hex',
defaultValue='#1c7ed6')
```

Description and placeholder

`description` adds helper text under the label, and `placeholder` fills the empty field.

Accent color

Source: Markdown

```
{% colorinput label='Accent color' description='Used for links and buttons'
placeholder='Pick a color' format='hex' %}
```

Source: Python

```
component('aardvark', 'colorinput',
          label='Accent color', description='Used for links and buttons',
          placeholder='Pick a color', format='hex')
```

Preset swatches

Pass `swatches` as a comma-separated list of colors to offer presets under the picker.

Pick a preset

Source: Markdown

```
{% colorinput label='Pick a preset' format='hex' swatches='#fa5252, #228be6, #40c057,
#fab005, #7950f2' %}
```

Source: Python

The `swatches` string is split on commas into the list Mantine expects:

```
component('aardvark', 'colorinput',
          label='Pick a preset', format='hex',
          swatches='#fa5252, #228be6, #40c057, #fab005, #7950f2')
```

Formats, pick-only, and sizes

The `format` controls how the value reads back — including the `*a` formats with alpha. Add `disallowInput` for a pick-only field (no typing), `disabled` for a read-only one, and `size` / `radius` (`xs`–`xl`) to scale it.

RGBA

Pick only

Disabled

Source: Markdown

```
{% colorinput label='RGBA' format='rgba' defaultValue='rgba(28, 126, 214, 0.6)' size='lg' %}
{% colorinput label='Pick only' format='hex' defaultValue='#40c057' disallowInput=true %}
{% colorinput label='Disabled' format='hex' defaultValue='#fab005' disabled=true %}
```

Source: Python

```
component('aardvark', 'colorinput', label='RGBA', format='rgba', defaultValue='rgba(28, 126,
214, 0.6)', size='lg')
component('aardvark', 'colorinput', label='Pick only', format='hex', defaultValue='#40c057',
disallowInput=True)
component('aardvark', 'colorinput', label='Disabled', format='hex', defaultValue='#fab005',
disabled=True)
```

With other components

Generate one color input per brand token from data with a Python loop — the same `component('aardvark', 'colorinput', …)` call, once per item, laid out in a card grid.

Primary

Success

Warning

Source: Python

```
tokens = [
    {"name": "Primary", "value": "#1c7ed6"},
    {"name": "Success", "value": "#40c057"},
    {"name": "Warning", "value": "#fab005"},
]
controls = ''
for t in tokens:
    field = component(
        'aardvark', 'colorinput',
        label=t["name"], format='hex', defaultValue=t["value"],
    )
    controls += component('aardvark', 'card', variant='plain', children=field)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=controls))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description
`label`	string	Field label.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder text for the empty field.
`defaultValue`	string	Starting color (matching the `format`).
`format`	`hex`, `hexa`, `rgb`, `rgba`, `hsl`, `hsla`	How the value reads back; the `*a` formats add an alpha channel.
`swatches`	comma-separated colors	Preset swatches shown under the picker (e.g. `'#fa5252, #228be6'`).

`size`	`xs–xl`	Field size.
`radius`	`xs–xl`	Corner rounding of the field.
`withPicker`	boolean	Show the dropdown picker (on by default; set `false` for swatch/text only).
`disallowInput`	boolean	Pick-only — disable typing into the field.
`withEyeDropper`	boolean	Show the eye-dropper button (on by default, where the browser supports it).
`disabled`	boolean	Make the field read-only.
`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

CSS Selectors

Target a `{% colorinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every ColorInput instance on the page */
[data-aardvark-island="ColorInput"] { }

/* Mantine Styles API parts */
.mantine-ColorInput-root { }
.mantine-ColorInput-input { }
.mantine-ColorInput-colorPreview { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Brand color

Source: Markdown

```
{% colorinput label='Brand color' format='hex' defaultValue='#1c7ed6' attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'colorinput', label='Brand color', format='hex',
defaultValue='#1c7ed6', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

ColorPicker

A **built-in** tag for an inline color picker — a saturation field plus a hue slider (and, for the alpha formats, an opacity slider), shown directly on the page rather than in a dropdown. It ships with Aardvark, so it's a single tag with no setup. Set a `format`, an optional `defaultValue`, and optional preset `swatches`.

Use it as `{% colorpicker %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'colorpicker', …)`.

Demonstrations

Basic picker

Set a `format` and an optional `defaultValue`. Drag in the saturation field and on the hue slider.

Source: Markdown

```
{% colorpicker format='hex' defaultValue='#1c7ed6' %}
```

Source: Python

```
component('aardvark', 'colorpicker', format='hex', defaultValue='#1c7ed6')
```

Alpha format and swatches

The `*a` formats (`rgba`, `hsla`, `hexa`) add an opacity slider. Pass `swatches` as a comma-separated list and `swatchesPerRow` to control the grid width.

Source: Markdown

```
{% colorpicker format='rgba' defaultValue='rgba(28, 126, 214, 0.8)' swatches='#fa5252,
#228be6, #40c057, #fab005, #7950f2' swatchesPerRow=5 %}
```

Source: Python

The `swatches` string is split on commas into the list Mantine expects:

```
component('aardvark', 'colorpicker',
          format='rgba', defaultValue='rgba(28, 126, 214, 0.8)',
          swatches='#fa5252, #228be6, #40c057, #fab005, #7950f2', swatchesPerRow=5)
```

Sizes and full width

`size` scales the control (`xs`–`xl`), and `fullWidth` stretches it to the container's width.

Source: Markdown

```
{% colorpicker format='hex' defaultValue='#7950f2' size='xl' fullWidth=true %}
```

Source: Python

```
component('aardvark', 'colorpicker', format='hex', defaultValue='#7950f2', size='xl',
fullWidth=True)
```

Swatches only

Turn the picker off with `withPicker=false` to offer a fixed palette and nothing else.

Source: Markdown

```
{% colorpicker format='hex' withPicker=false swatches='#fa5252, #228be6, #40c057, #fab005,
#7950f2' %}
```

Source: Python

```
component('aardvark', 'colorpicker',
          format='hex', withPicker=False,
          swatches='#fa5252, #228be6, #40c057, #fab005, #7950f2')
```

With other components

Generate one swatches-only picker per palette from data with a Python loop — the same `component('aardvark', 'colorpicker', …)` call, once per item, laid out in a card grid.

Warm

Cool

Royal

Source: Python

```
palettes = [
    {"name": "Warm", "colors": "#fa5252, #fab005, #fd7e14"},
    {"name": "Cool", "colors": "#228be6, #15aabf, #40c057"},
    {"name": "Royal", "colors": "#7950f2, #be4bdb, #e64980"},
]
```

```
controls = ''
for p in palettes:
    picker = '**' + p["name"] + '**\n\n' + component(
        'aardvark', 'colorpicker',
        format='hex', withPicker=False, swatches=p["colors"], swatchesPerRow=3,
    )
    controls += component('aardvark', 'card', variant='plain', children=picker)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=controls))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description
`defaultValue`	string	Starting color (matching the `format`).
`format`	`hex`, `hexa`, `rgb`, `rgba`, `hsl`, `hsla`	Color format; the `*a` formats add the alpha (opacity) slider.
`swatches`	comma-separated colors	Preset swatches below the picker (e.g. `'#fa5252, #228be6'`).
`swatchesPerRow`	integer	How many swatches per row.
`size`	`xs`–`xl`	Size of the control.
`fullWidth`	boolean	Stretch the control to the container's width.
`withPicker`	boolean	Show the saturation field and sliders (on by default; set `false` to show only swatches).
`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

CSS Selectors

Target a `{% colorpicker %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every ColorPicker instance on the page */
[data-aardvark-island="ColorPicker"] { }

/* Mantine Styles API parts */
.mantine-ColorPicker-wrapper { }
.mantine-ColorPicker-saturation { }
.mantine-ColorPicker-slider { }
```

```
.mantine-ColorPicker-swatch { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the picker reads the changed control value, logs it to the console, and alerts it:

Source: Markdown

```
{% colorpicker format='hex' defaultValue='#1c7ed6' attr={'onchange': '''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'colorpicker', format='hex', defaultValue='#1c7ed6', attr={'onchange':
'''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''})
```

DateInput

A **free-typing date field**: type a date (or pick it) and it's parsed against `valueFormat`, showing the formatted result. Reach for it when a plain, keyboard-first date entry beats a full calendar popover. It carries the same Input wrapper as TextInput — label, description, error, sizes — and hydrates into an interactive island.

Use it as `{% dateinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'dateinput', …)`.

Dates are strings

Mantine 9's date components take and return **date strings** — `YYYY-MM-DD` (or `YYYY-MM-DD HH:mm` when the format includes time), never `Date` objects. So `defaultValue` is a string like `'2026-01-15'`.

A basic field

`defaultValue` seeds the field (a `YYYY-MM-DD` string); `valueFormat` controls how it's displayed.

Release date

Source: Markdown

```
{% dateinput label='Release date' defaultValue='2026-01-15' valueFormat='MMM D, YYYY'
clearable=true %}
```

Source: Python

```
component('aardvark', 'dateinput', label='Release date',
          defaultValue='2026-01-15', valueFormat='MMM D, YYYY', clearable=True)
```

Sizes, variants, and validation

`size` and `radius` run `xs`–`xl`; `variant` is `default`, `filled`, or `unstyled`. `error` shows a validation message, and `required` / `withAsterisk` add the asterisk.

Start

Due

Source: Markdown

```
{% dateinput label='Start' description='Pick the first day' size='md' variant='filled'
required=true placeholder='Choose a date' %}

{% dateinput label='Due' error='A due date is required' %}
```

Source: Python

```
component('aardvark', 'dateinput', label='Start', description='Pick the first day',
          size='md', variant='filled', required=True, placeholder='Choose a date')

component('aardvark', 'dateinput', label='Due', error='A due date is required')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`defaultValue`	`YYYY-MM-DD` string	Initial value (a date string, never a `Date`).
`valueFormat`	a dayjs format string (e.g. `MMM D, YYYY`)	How the value is displayed and parsed.
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when empty.
`error`	string	Validation message; switches the field to the error color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`–`xl` or a CSS length	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`required`	bool (`true` / `false`)	Mark required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required`.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`clearable`	bool (`true` / `false`)	Show an × to clear the value.

CSS Selectors

Target a `{% dateinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every DateInput instance on the page */
[data-aardvark-island="DateInput"] { }

/* Mantine Styles API parts */
.mantine-DateInput-root { }
.mantine-DateInput-input { }
.mantine-DateInput-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so committing a typed value or picking a date logs the new value to the console and alerts it:

Release date

Source: Markdown

```
{% dateinput label='Release date' defaultValue='2026-01-15' valueFormat='MMM D, YYYY'
clearable=true attr={'onchange': '''
const value = event.target.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'dateinput', label='Release date',
          defaultValue='2026-01-15', valueFormat='MMM D, YYYY', clearable=True,
attr={'onchange': '''
const value = event.target.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

DatePickerInput

A date field that **opens a calendar popover** to pick a value — one day, a **range**, or **several** days, set by `type`. Reach for it over DateInput when you want the calendar UI rather than free typing. It carries the usual Input wrapper and hydrates into an interactive island.

Use it as `{% datepickerinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'datepickerinput', …)`.

Dates are strings

Values are **date strings** (`YYYY-MM-DD`). For `type='range'` or `type='multiple'`, pass `defaultValue` as a **JSON array** of those strings — e.g. `'["2026-01-01","2026-01-07"]'`.

Pick a single day

Ship date

Source: Markdown

```
{% datepickerinput label='Ship date' defaultValue='2026-03-09' valueFormat='MMM D, YYYY'
clearable=true %}
```

Source: Python

```
component('aardvark', 'datepickerinput', label='Ship date',
          defaultValue='2026-03-09', valueFormat='MMM D, YYYY', clearable=True)
```

Pick a range

`type='range'` selects a start and end day; seed it with a two-element JSON array.

Sprint

Source: Markdown

```
{% datepickerinput label='Sprint' type='range' defaultValue='["2026-01-05","2026-01-16"]' %}
```

Source: Python

```
component('aardvark', 'datepickerinput', label='Sprint', type='range',
          defaultValue='["2026-01-05","2026-01-16"]')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`type`	`default` (single), `range`, `multiple`	What the calendar selects.
`defaultValue`	`YYYY-MM-DD` string, or a JSON array of them for `range` / `multiple`	Initial value.
`valueFormat`	a dayjs format string	How the value is displayed.
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when empty.
`error`	string	Validation message; switches the field to the error color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`–`xl` or a CSS length	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`required`	bool (`true` / `false`)	Mark required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required`.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`clearable`	bool (`true` / `false`)	Show an × to clear the value.

CSS Selectors

Target a `{% datepickerinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every DatePickerInput instance on the page */
[data-aardvark-island="DatePickerInput"] { }

/* Mantine Styles API parts */
.mantine-DatePickerInput-root { }
.mantine-DatePickerInput-input { }
```

```
.mantine-DatePickerInput-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so choosing a date logs the field value to the console and alerts it:

Ship date

Source: Markdown

```
{% datepickerinput label='Ship date' defaultValue='2026-03-09' valueFormat='MMM D, YYYY'
clearable=true attr={'onchange': '''
const value = event.target.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'datepickerinput', label='Ship date',
          defaultValue='2026-03-09', valueFormat='MMM D, YYYY', clearable=True,
attr={'onchange': '''
const value = event.target.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

DateTimePicker

A combined **date-and-time field** — it opens a calendar to pick the day and a time picker for the clock, in one control. Reach for it when an event needs both. It carries the usual Input wrapper and hydrates into an interactive island.

Use it as `{% datetimepicker %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'datetimepicker', …)`.

Dates are strings

The value is a **date-time string** — `YYYY-MM-DD HH:mm:ss` (or `YYYY-MM-DD HH:mm` without `withSeconds`), never a `Date`.

A date-and-time field

`withSeconds` adds the seconds field; `valueFormat` controls the display.

Starts at

Source: Markdown

```
{% datetimepicker label='Starts at' defaultValue='2026-01-15 09:30:00' withSeconds=true
valueFormat='MMM D, YYYY HH:mm:ss' clearable=true %}
```

Source: Python

```
component('aardvark', 'datetimepicker', label='Starts at',
          defaultValue='2026-01-15 09:30:00', withSeconds=True,
          valueFormat='MMM D, YYYY HH:mm:ss', clearable=True)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`defaultValue`	`YYYY-MM-DD HH:mm:ss` string	Initial value (a date-time string).
`valueFormat`	a dayjs format string	How the value is displayed.
`withSeconds`	bool (`true` / `false`)	Show and capture seconds.
`label`	string	Field label above the input.

`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when empty.
`error`	string	Validation message; switches the field to the error color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`–`xl` or a CSS length	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`required`	bool (`true` / `false`)	Mark required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required`.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`clearable`	bool (`true` / `false`)	Show an × to clear the value.

CSS Selectors

Target a `{% datetimepicker %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every DateTimePicker instance on the page */
[data-aardvark-island="DateTimePicker"] { }

/* Mantine Styles API parts */
.mantine-DateTimePicker-root { }
.mantine-DateTimePicker-input { }
.mantine-DateTimePicker-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so choosing a date-time logs the field value to the console and alerts it:

Starts at

Source: Markdown

```
{% datetimepicker label='Starts at' defaultValue='2026-01-15 09:30:00' withSeconds=true
valueFormat='MMM D, YYYY HH:mm:ss' clearable=true attr={'onchange': '''
const value = event.target.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'datetimepicker', label='Starts at',
          defaultValue='2026-01-15 09:30:00', withSeconds=True,
          valueFormat='MMM D, YYYY HH:mm:ss', clearable=True, attr={'onchange': '''
const value = event.target.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

Dropzone

A **drag-and-drop file upload zone**: drop files onto it, or click it to open the native file picker. It shows a built-in drag-state overlay (an accept/reject tint as files hover) and hydrates into an interactive island. Reach for it when a file field needs a generous drop target rather than the compact FileInput button.

Restrict what it takes with `accept` (a comma-separated MIME list or a JSON array), cap file size with `maxSize` (bytes), and toggle `multiple`, `loading`, and `disabled`. The block body is the zone's resting content — the prompt a visitor sees before they drag anything in. After a file is dropped or selected, the zone shows the selected file names.

Use it as `{% dropzone %}…{% endDropzone %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'dropzone', …)`.

A basic zone

The block body is the prompt; `accept` and `maxSize` constrain what may be dropped.

Drag images here or click to choose — PNG or JPEG, up to 5 MB.

Source: Markdown

```
{% dropzone accept='image/png,image/jpeg' maxSize=5242880 h='180' %}
**Drag images here** or click to choose — PNG or JPEG, up to 5 MB.
{% endDropzone %}
```

Source: Python

```
component('aardvark', 'dropzone',
          accept='image/png,image/jpeg', maxSize=5242880, h='180',
          children='**Drag images here** or click to choose — PNG or JPEG, up to 5 MB.')
```

Single file, disabled, and loading

`multiple` is on by default; set `multiple=false` for a one-file zone. `loading` overlays a spinner and `disabled` greys the zone out and stops it capturing files.

Drop a single **PDF**, or click to browse.

Uploads are paused right now.

Source: Markdown

```
{% dropzone multiple=false accept='application/pdf' %}
Drop a single **PDF**, or click to browse.
{% endDropzone %}
```

```
{% dropzone disabled=true %}
Uploads are paused right now.
{% endDropzone %}
```

Source: Python

```
component('aardvark', 'dropzone', multiple=False, accept='application/pdf',
          children='Drop a single **PDF**, or click to browse.')

component('aardvark', 'dropzone', disabled=True,
          children='Uploads are paused right now.')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`accept`	MIME list (`image/png,image/jpeg`) or a JSON array	Restrict accepted file types; omit to accept any file.
`maxSize`	integer (bytes)	Maximum size of a single file.
`multiple`	bool (`true` / `false`)	Allow several files at once (default `true`).
`loading`	bool (`true` / `false`)	Show a loading overlay over the zone.
`disabled`	bool (`true` / `false`)	Disable file capturing and grey the zone out.
`radius`	`xs`–`xl` or a CSS length	Corner radius.
`h`	number or CSS length	Height of the zone.
`minHeight`	number or CSS length	Minimum height of the zone.
`maw`	number or CSS length	Maximum width of the zone.
`name`	string	Form control name submitted with the value.

CSS Selectors

The tag mounts as an island, so its rendered markup carries the island's data attributes plus Mantine's own component classes — target either to restyle a zone.

```
/* Every dropzone on the page (the island mount element) */
[data-aardvark-island="Dropzone"] {
  border-style: dashed;
}

/* Mantine's own classes on the hydrated element */
.mantine-Dropzone-root {
  background: var(--mantine-color-gray-0);
}
.mantine-Dropzone-inner {
  padding: 2rem;
}

/* Persistent selected/rejected file summary */
.aardvark-Dropzone-status { }
```

Injecting Attributes

Pass an `attr` map to set raw HTML attributes (or inline handlers) on the zone's rendered root element. Here it is wired to `onchange`; selecting files from the picker or dropping files into the zone logs the selected file names to the console and alerts them:

Drop files here

Source: Markdown

```
{% dropzone attr={'onchange': '''
const value = Array.from(event.target.files || []).map((file) => file.name).join(', ') ||
event.target.value;
console.log('attr demo value:', value);
alert(value || 'No files selected');
'''} %}Drop files here{% endDropzone %}
```

Source: Python

```
component('aardvark', 'dropzone', children='Drop files here', attr={'onchange': '''
const value = Array.from(event.target.files || []).map((file) => file.name).join(', ') ||
event.target.value;
console.log('attr demo value:', value);
alert(value || 'No files selected');
'''})
```

Fieldset

A built-in block tag that groups related fields in a bordered box with a `legend` caption. The block body holds the inputs, and `disabled` can cascade a disabled state to every control inside. Use it to break a long form into labelled sections.

Use it as `{% fieldset %} … {% endFieldset %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'fieldset', …)` with the grouped fields passed as `children`.

A labelled group

The body between `{% fieldset %}` and `{% endFieldset %}` holds the fields; `legend` is the caption rendered in the border.

First name

Last name

Source: Markdown

```
{% fieldset legend='Personal information' %}
{% textinput label='First name' placeholder='Ada' %}
{% textinput label='Last name' placeholder='Lovelace' %}
{% endFieldset %}
```

Source: Python

```
inner = (
    component('aardvark', 'textinput', label='First name', placeholder='Ada')
    + component('aardvark', 'textinput', label='Last name', placeholder='Lovelace')
)
component('aardvark', 'fieldset', legend='Personal information', children=inner)
```

Variant

`variant` is `default`, `filled`, or `unstyled`.

Email

Password

Source: Markdown

```
{% fieldset legend='Account (filled)' variant='filled' %}
{% textinput label='Email' placeholder='you@example.com' %}
{% passwordinput label='Password' %}
{% endFieldset %}
```

Source: Python

```
inner = (
    component('aardvark', 'textinput', label='Email', placeholder='you@example.com')
    + component('aardvark', 'passwordinput', label='Password')
)
component('aardvark', 'fieldset', legend='Account (filled)', variant='filled',
children=inner)
```

Radius

`radius` sets the border radius — `xs`–`xl` or any CSS value.

Nickname

Source: Markdown

```
{% fieldset legend='Rounded' radius='lg' %}
{% textinput label='Nickname' placeholder='Ada' %}
{% endFieldset %}
```

Source: Python

```
inner = component('aardvark', 'textinput', label='Nickname', placeholder='Ada')
component('aardvark', 'fieldset', legend='Rounded', radius='lg', children=inner)
```

Disabled cascade

`disabled` cascades a disabled state to every control inside the group, so you can lock a whole section with one flag.

Read-only field

Plan

Source: Markdown

```
{% fieldset legend='Locked' disabled=true %}
{% textinput label='Read-only field' defaultValue='Cannot edit' %}
{% nativeselect label='Plan' data='Free|Pro' %}
{% endFieldset %}
```

Source: Python

```
inner = (
    component('aardvark', 'textinput', label='Read-only field', defaultValue='Cannot edit')
    + component('aardvark', 'nativeselect', label='Plan', data='Free|Pro')
)
component('aardvark', 'fieldset', legend='Locked', disabled=True, children=inner)
```

With other components

Fieldset is a container, so any of the field tags can go inside it — text inputs, selects, file pickers, and more — to compose a complete form section.

Street address

Country

Delivery note (optional)

Source: Markdown

```
{% fieldset legend='Shipping' %}
{% textinput label='Street address' placeholder='123 Analytical Ave' %}
{% nativeselect label='Country' data='United States|Canada|United Kingdom' %}
{% fileinput label='Delivery note (optional)' placeholder='Attach a file' %}
{% endFieldset %}
```

Source: Python

```
inner = (
    component('aardvark', 'textinput', label='Street address', placeholder='123 Analytical
Ave')
    + component('aardvark', 'nativeselect', label='Country', data='United
States|Canada|United Kingdom')
    + component('aardvark', 'fileinput', label='Delivery note (optional)',
placeholder='Attach a file')
)
component('aardvark', 'fieldset', legend='Shipping', children=inner)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`legend`	string	The caption rendered in the border.
`variant`	`default`, `filled`, `unstyled`	Visual style of the bordered group.
`radius`	`xs`, `sm`, `md`, `lg`, `xl` (or any CSS value)	Border radius.
`disabled`	`true` / `false`	Disable every control inside the group. Default `false`.

(body)	Markdown / field tags	The grouped inputs. In Python, pass them as `children`.

CSS Selectors

Target a `{% fieldset %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Fieldset instance on the page */
[data-aardvark-island="Fieldset"] { }

/* Mantine Styles API parts */
.mantine-Fieldset-root { }
.mantine-Fieldset-legend { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so editing the nested field logs the field value to the console and alerts it:

Name

Source: Markdown

```
{% fieldset legend='Personal information' attr={'onchange': '''
const field = event.target.closest('input, textarea, select');
const value = field ? field.value : '';
console.log('attr demo value:', value);
alert(value);
'''} %}{% textinput label='Name' placeholder='Ada Lovelace' %}{% endFieldset %}
```

Source: Python

```
component('aardvark', 'fieldset', legend='Personal information',
          children=component('aardvark', 'textinput', label='Name', placeholder='Ada
Lovelace'),
          attr={'onchange': '''
const field = event.target.closest('input, textarea, select');
const value = field ? field.value : '';
console.log('attr demo value:', value);
alert(value);
'''})
```

FileInput

A built-in tag for a file picker that matches the rest of your form. It uses the same Input wrapper as TextInput, so it carries a label, description, placeholder, and error message, and adds the file-specific options `accept`, `multiple`, and `clearable`.

Use it as `{% fileinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'fileinput', …)`.

Basic file picker

A labelled picker with a placeholder shown until a file is chosen.

Resume

Source: Markdown

```
{% fileinput label='Resume' placeholder='Choose a file' %}
```

Source: Python

```
component('aardvark', 'fileinput', label='Resume', placeholder='Choose a file')
```

Accept, multiple, and clearable

`accept` is the native accept-attribute string; `multiple` lets the picker take several files; `clearable` shows a clear button once a file is chosen.

Images

Source: Markdown

```
{% fileinput label='Images' accept='image/png,image/jpeg' multiple=true clearable=true
placeholder='Pick one or more images' %}
```

Source: Python

```
component(
    'aardvark', 'fileinput',
    label='Images',
    accept='image/png,image/jpeg',
    multiple=True,
    clearable=True,
    placeholder='Pick one or more images',
)
```

Description, required, and error

The shared wrapper fields work as on the other inputs: `description` adds help text, `required` (or `withAsterisk`) adds the asterisk, and `error` shows a validation message and switches the field to the error color.

Contract

Source: Markdown

```
{% fileinput label='Contract' description='PDF only' accept='application/pdf' required=true
error='A signed contract is required' %}
```

Source: Python

```
component(
    'aardvark', 'fileinput',
    label='Contract',
    description='PDF only',
    accept='application/pdf',
    required=True,
    error='A signed contract is required',
)
```

Variant, size, and radius

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`.

Filled

Large + round

Source: Markdown

```
{% fileinput label='Filled' variant='filled' placeholder='filled variant' %}

{% fileinput label='Large + round' size='lg' radius='xl' placeholder='lg / xl radius' %}
```

Source: Python

```
component('aardvark', 'fileinput', label='Filled', variant='filled', placeholder='filled
variant')

component('aardvark', 'fileinput', label='Large + round', size='lg', radius='xl',
placeholder='lg / xl radius')
```

Sections and disabled

`leftSection` and `rightSection` put text inside the field; `disabled` renders it inert.

Attachment

Locked

Source: Markdown

```
{% fileinput label='Attachment' leftSection='⎕' placeholder='Add a file' %}

{% fileinput label='Locked' disabled=true placeholder='Upload disabled' %}
```

Source: Python

```
component('aardvark', 'fileinput', label='Attachment', leftSection='⎕', placeholder='Add a
file')

component('aardvark', 'fileinput', label='Locked', disabled=True, placeholder='Upload
disabled')
```

With other components

Group a file picker with other fields inside a fieldset.

Full name

Resume

Source: Markdown

```
{% fieldset legend='Application' %}
{% textinput label='Full name' placeholder='Ada Lovelace' %}
{% fileinput label='Resume' accept='application/pdf' placeholder='Upload your resume' %}
{% endFieldset %}
```

Source: Python

```
inner = (
    component('aardvark', 'textinput', label='Full name', placeholder='Ada Lovelace')
    + component('aardvark', 'fileinput', label='Resume', accept='application/pdf',
placeholder='Upload your resume')
)
component('aardvark', 'fieldset', legend='Application', children=inner)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	Label shown above the field.
`description`	string	Help text shown under the label.
`placeholder`	string	Placeholder shown until a file is chosen.
`error`	string	Validation message shown below the field; switches it to the error color.
`accept`	string	Native accept-attribute string, e.g. `image/png,image/jpeg` or `application/pdf`.
`multiple`	`true` / `false`	Allow selecting several files. Default `false`.
`clearable`	`true` / `false`	Show a clear button once a file is chosen. Default `false`.
`variant`	`default`, `filled`, `unstyled`	Visual style.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Field size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl` (or any CSS value)	Corner radius.
`required`	`true` / `false`	Mark the field required and add the asterisk. Default `false`.
`withAsterisk`	`true` / `false`	Add the asterisk without the `required` semantics. Default `false`.
`disabled`	`true` / `false`	Render the field disabled. Default `false`.
`leftSection`	string	Text shown inside the field on the left.
`rightSection`	string	Text shown inside the field on the right.

CSS Selectors

Target a `{% fileinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every FileInput instance on the page */
```

```
[data-aardvark-island="FileInput"] { }

/* Mantine Styles API parts */
.mantine-FileInput-root { }
.mantine-FileInput-input { }
.mantine-FileInput-placeholder { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so choosing a file logs the selected file name to the console and alerts it:

Resume

Source: Markdown

```
{% fileinput label='Resume' placeholder='Choose a file' attr={'onchange': '''
const value = Array.from(event.target.files || []).map((file) => file.name).join(', ');
console.log('attr demo value:', value);
alert(value || 'No file selected');
'''} %}
```

Source: Python

```
component('aardvark', 'fileinput', label='Resume', placeholder='Choose a file',
attr={'onchange': '''
const value = Array.from(event.target.files || []).map((file) => file.name).join(', ');
console.log('attr demo value:', value);
alert(value || 'No file selected');
'''})
```

FocusTrap

`focustrap` is a **focus-management behavior primitive**. While it's `active`, it traps keyboard focus inside its content: `Tab` and `Shift+Tab` cycle through the focusable elements within the region instead of leaving it. It's the focus behavior that powers modals and drawers, so a keyboard user can't tab "behind" an open dialog.

It renders **no visual chrome of its own**. The trapped child supplies the panel, form, or overlay you want readers to see.

Use it as `{% focustrap %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'focustrap', …)`. Close the block with `{% endFocustrap %}` (one capital F).

Demonstrations

Activate the demo, then press `Tab`. Focus moves into the trap and cycles through the field and buttons until you release it. With the trap off, `Tab` can reach the outside control.

Wrap a region whose focusable elements should be trapped; the block body is that region. In a real modal, mount the trap only while the overlay is open, or pass an `active` value from a stateful island:

secondthird

Set `active=false` to keep the region rendered while the focus trap is disengaged.

Source: Live demo

```
{% FocusTrapDemo %}
```

Source: Markdown

```
{% focustrap %}
<div style="display:flex; gap:0.5rem; align-items:center;">
  <input placeholder="first" />
  <button>second</button>
  <a href="#">third</a>
</div>
{% endFocustrap %}
```

Source: Python

```
region = (
    '<div style="display:flex; gap:0.5rem; align-items:center;">'
    '<input placeholder="first" />'
    '<button>second</button>'
    '<a href="#">third</a>'
    '</div>'
```

```
)
component('aardvark', 'focustrap', children=region)
```

Use `active=false` to disable the trap without removing it. This is useful when a parent owns the open/closed state and only wants focus captured some of the time:

Source: Markdown

```
{% focustrap active=false %}
<div>
  <input placeholder="first" />
  <button>second</button>
</div>
{% endFocustrap %}
```

Source: Python

```
component(
    'aardvark', 'focustrap',
    active=False,
    children='<div><input placeholder="first" /><button>second</button></div>',
)
```

With other components

A trap wraps the focusable region of a custom overlay you build — for example the body of a `paper` panel acting as a dialog. This rendered example is disengaged so it does not capture page focus while you read:

Subscribe

Source: Markdown

```
{% paper withBorder=true shadow='md' radius='md' p='lg' %}
{% focustrap %}
<div style="display:flex; gap:0.5rem; align-items:center;"><input placeholder="email"
/><button>Subscribe</button></div>
{% endFocustrap %}
{% endPaper %}
```

Source: Python

```
form = component(
    'aardvark', 'focustrap',
    children='<div style="display:flex; gap:0.5rem; align-items:center;">'
             '<input placeholder="email" /><button>Subscribe</button></div>',
)
component('aardvark', 'paper', withBorder=True, shadow='md', radius='md', p='lg',
          children=form)
```

Attributes

Attribute	Valid values	Description
`active`	bare flag or `true` / `false` (default `true`)	Whether the trap is engaged. Set `active=false` to disable it without removing it.

`attr={...}` forwards raw HTML attributes onto the rendered element.

CSS Selectors

Target FocusTrap from your own CSS with the island data attribute. FocusTrap is a behavior wrapper, so it renders no Mantine part classes of its own:

```
/* Every FocusTrap instance on the page */
[data-aardvark-island="FocusTrap"] { }

/* FocusTrap is a behavior wrapper — it renders no .mantine-FocusTrap-* parts,
   forwarding its markup through unchanged. Style its data attribute instead. */
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — onto the trapped element. FocusTrap is not itself a value input, so this example makes a `nativeselect` the trapped child and attaches `onchange` to that input. Choose an option: the handler reads the selected value, writes it back into the demo, logs it, and alerts it.

Framework

Choose an option.

Source: Markdown

```
<div data-focustrap-attr-demo>
{% focustrap active=false attr={'onchange': '''
const value = event.target.value || '';
const root = event.target.closest('[data-focustrap-attr-demo]');
const output = root ? root.querySelector('[data-focustrap-output]') : null;
if (output) output.textContent = value ? 'Last value: ' + value : 'No value entered';
console.log('focustrap attr value:', value);
alert(value);
'''} %}
{% nativeselect label='Framework' data='React|Vue|Svelte|Angular' %}
{% endFocustrap %}
<small data-focustrap-output>Choose an option.</small>
</div>
```

Source: Python

```
handler = '''
const value = event.target.value || '';
const root = event.target.closest('[data-focustrap-attr-demo]');
const output = root ? root.querySelector('[data-focustrap-output]') : null;
if (output) output.textContent = value ? 'Last value: ' + value : 'No value entered';
console.log('focustrap attr value:', value);
alert(value);
'''

field = component(
    'aardvark', 'focustrap',
    active=False,
    children=component('aardvark', 'nativeselect',
                       label='Framework', data='React|Vue|Svelte|Angular'),
    attr={'onchange': handler},
)

page.print(
    '<div data-focustrap-attr-demo>'
    + field
    + '<small data-focustrap-output>Choose an option.</small>'
    + '</div>'
)
```

Form

`useForm` is the hook for managing form state: it holds the field values, tracks which fields have been touched or changed, runs validation, and produces the props each input needs. You connect a field to the form by spreading `getInputProps('name')` onto an input — that wires up its value, `onChange`, `onBlur`, current error, and the key the hook uses to address the field.

Submitting goes through `onSubmit(handler)`: it runs every validation rule first, shows the error messages on any field that fails, and only calls your handler — with the collected values — once the whole form is valid. That gives you controlled inputs, inline validation, and a clean submit path without hand-wiring `useState` for every field.

Because `useForm` is a hook with no component of its own, this page ships a worked example as the `{% ContactForm %}` snippet: three fields (name, email, message), one plain validation rule each, and a result area that prints the values when the form passes.

Live example

Submit with empty or invalid fields to see the inline errors; fill everything in and the collected values appear below the buttons.

Source: Markdown

```
{% ContactForm %}
```

Source: snippet

```
import { forwardRef, useState } from 'react';
import { useForm } from '@mantine/form';
import { Button, Group, Stack, TextInput } from '@mantine/core';

const ContactForm = forwardRef(function ContactForm(props, ref) {
  const form = useForm({
    initialValues: { name: '', email: '', message: '' },
    validate: {
      email: (value) => (/^\S+@\S+\.\S+$/.test(value) ? null : 'Enter a valid email
address'),
    },
  });
  return (
    <form ref={ref} onSubmit={form.onSubmit((values) => console.log(values))} {...props}>
      <Stack gap="sm">
        <TextInput label="Email" key={form.key('email')} {...form.getInputProps('email')} />
        <Group><Button type="submit">Send</Button></Group>
      </Stack>
    </form>
  );
});
```

What the demo shows

Concept	How it appears in the demo
Field binding	Each input spreads `getInputProps('name')`, so its value and error come from the form.
Validation	One plain rule per field (length / email pattern); a rule returns a string to flag an error or `null` to pass.
Submit handling	`onSubmit(handler)` validates first, then prints the collected values into the result area.
Reset	The Reset button calls `form.reset()` to restore the initial values and clear errors.
Inline errors	A failed rule renders its message under the field via the input's error slot.

CSS Selectors

The snippet renders a single `<form>` element carrying the island marker, with Mantine inputs inside it. Target the form itself through the island attribute, and the input parts through the Styles API class names Mantine emits for `TextInput` / `Textarea`:

```
/* the outer <form> the snippet renders */
[data-aardvark-island="ContactForm"] {
  max-width: 32rem;
}

/* the text field control inside each input */
[data-aardvark-island="ContactForm"] .mantine-TextInput-input {
  font-variant-numeric: tabular-nums;
}

/* the validation message shown under a failing field */
[data-aardvark-island="ContactForm"] .mantine-InputWrapper-error {
  font-style: italic;
}
```

Injecting Attributes

The `attr` prop forwards raw HTML attributes onto the form's root DOM node — the snippet forwards its `ref`, so anything you pass lands on the outer `<form>` element. Use it for event handlers, `data-*` hooks, or ARIA attributes the snippet doesn't expose directly. Here it is wired to `onchange`, so changing any field logs that field's current value to the console and alerts it:

Source: Markdown

```
{% ContactForm attr={'onchange': '''
const field = event.target.closest('input, textarea, select');
const value = field ? field.value : '';
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('ContactForm', attr={'onchange': '''
const field = event.target.closest('input, textarea, select');
const value = field ? field.value : '';
console.log('attr demo value:', value);
alert(value);
'''})
```

HueSlider

A **built-in** tag for a hue slider — a thin track of the full color spectrum for picking a hue from 0 to 360. It's one of Mantine's color-picker building blocks. It ships with Aardvark, so it's a single tag with no setup. Set `defaultValue` for the starting hue, and tune the track with `size` and `radius`.

Use it as `{% hueslider %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'hueslider', …)`.

Demonstrations

Basic hue

Set a `defaultValue` from 0 to 360 (the starting hue). Drag along the spectrum to change it.

Source: Markdown

```
{% hueslider defaultValue=120 %}
```

Source: Python

```
component('aardvark', 'hueslider', defaultValue=120)
```

Sizes and radius

`size` sets the track thickness (`xs`–`xl`) and `radius` rounds its ends (`xs`–`xl`).

Source: Markdown

```
{% hueslider defaultValue=0 size='xl' radius='xl' %}
{% hueslider defaultValue=200 size='sm' %}
```

Source: Python

```
component('aardvark', 'hueslider', defaultValue=0, size='xl', radius='xl')
component('aardvark', 'hueslider', defaultValue=200, size='sm')
```

With other components

Generate one hue slider per channel from data with a Python loop — the same `component('aardvark', 'hueslider', …)` call, once per item, laid out in a card grid.

Red zone

Green zone

Blue zone

Source: Python

```
channels = [
    {"name": "Red zone", "hue": 0},
    {"name": "Green zone", "hue": 120},
    {"name": "Blue zone", "hue": 240},
]
controls = ''
for c in channels:
    row = '**' + c["name"] + '**\n\n' + component(
        'aardvark', 'hueslider', defaultValue=c["hue"], size='lg',
    )
    controls += component('aardvark', 'card', variant='plain', children=row)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=controls))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description
`defaultValue`	integer `0–360`	Starting hue — the reader can drag it.
`size`	`xs–xl`	Track thickness.
`radius`	`xs–xl`	Corner rounding of the track ends.
`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

CSS Selectors

Target a `{% hueslider %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every HueSlider instance on the page */
[data-aardvark-island="HueSlider"] { }

/* Mantine Styles API parts */
.mantine-HueSlider-slider { }
```

```
.mantine-HueSlider-sliderOverlay { }
.mantine-HueSlider-thumb { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so dragging the hue control reads the changed control value, logs it to the console, and alerts it:

Source: Markdown

```
{% hueslider defaultValue=120 attr={'onchange': '''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'hueslider', defaultValue=120, attr={'onchange': '''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''})
```

Input

A built-in tag for the base input primitive — the styled control with no label, description, or error message of its own. Every other field on this page is built on it. Reach for `input` when you want just the box (for example, inside your own layout). For a full labelled field, use TextInput instead.

Use it as `{% input %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'input', …)`.

No wrapper — just the control

`Input` is the unstyled base. The label, description, and error messages you see on the other fields come from `Input.Wrapper`, which they compose on top of `Input`. So `input` here has no `label` or `description`, and its `error` is a **boolean** that adds error styling rather than a message string.

Basic control

`placeholder` is shown when the control is empty; `type` sets the HTML input type.

Source: Markdown

```
{% input placeholder='Search…' type='search' %}
```

Source: Python

```
component('aardvark', 'input', placeholder='Search…', type='search')
```

Type and default value

`type` accepts any HTML input type (`search`, `email`, `tel`, …); `defaultValue` sets the initial value.

Source: Markdown

```
{% input type='email' placeholder='you@example.com' %}

{% input type='tel' defaultValue='+1 555 0100' %}
```

Source: Python

```
component('aardvark', 'input', type='email', placeholder='you@example.com')

component('aardvark', 'input', type='tel', defaultValue='+1 555 0100')
```

Variant, size, and radius

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`.

Source: Markdown

```
{% input placeholder='Filled' variant='filled' %}

{% input placeholder='Large + round' size='lg' radius='xl' %}
```

Source: Python

```
component('aardvark', 'input', placeholder='Filled', variant='filled')

component('aardvark', 'input', placeholder='Large + round', size='lg', radius='xl')
```

Error and disabled

`error` is a boolean that adds error styling (there is no wrapper to render a message); `disabled` renders the control inert.

Source: Markdown

```
{% input placeholder='Error styling' error=true %}

{% input placeholder='Disabled' disabled=true %}
```

Source: Python

```
component('aardvark', 'input', placeholder='Error styling', error=True)

component('aardvark', 'input', placeholder='Disabled', disabled=True)
```

Sections

`leftSection` and `rightSection` put text inside the control — handy for units or symbols.

Source: Markdown

```
{% input placeholder='Amount' leftSection='$' rightSection='USD' %}
```

Source: Python

```
component('aardvark', 'input', placeholder='Amount', leftSection='$', rightSection='USD')
```

With other components

`input` has no label, so when you need an accessible field reach for the wrapped textinput instead. The bare control is best for composing your own layouts — here it sits next to a button as an inline search bar inside a group.

Search

Source: Markdown

```
{% group %}
{% input placeholder='Search the docs…' type='search' %}
{% button %}Search{% endButton %}
{% endGroup %}
```

Source: Python

```
inner = (
    component('aardvark', 'input', placeholder='Search the docs…', type='search')
    + component('aardvark', 'button', children='Search')
)
component('aardvark', 'group', children=inner)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`placeholder`	string	Placeholder shown when the control is empty.
`type`	HTML input type (`text`, `search`, `email`, `tel`, `url`, …)	The native input type.
`defaultValue`	string	Initial value of the control.
`variant`	`default`, `filled`, `unstyled`	Visual style.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl` (or any CSS value)	Corner radius.
`disabled`	`true` / `false`	Render the control disabled. Default `false`.

`error`	`true` / `false`	Add error styling. Boolean — there is no wrapper for a message. Default `false`.
`leftSection`	string	Text shown inside the control on the left.
`rightSection`	string	Text shown inside the control on the right.

CSS Selectors

Target a `{% input %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Input instance on the page */
[data-aardvark-island="Input"] { }

/* Mantine Styles API parts */
.mantine-Input-wrapper { }
.mantine-Input-input { }
.mantine-Input-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Source: Markdown

```
{% input placeholder='Search…' type='search' attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'input', placeholder='Search…', type='search', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

JsonInput

A JSON-aware textarea: it validates the content as JSON and can reformat it on blur. It carries the same Input wrapper as TextInput and the multi-line behaviour of Textarea, plus the JSON specifics: `formatOnBlur` (pretty-print on blur) and `validationError` (the message shown for invalid JSON). Reach for it for config blobs, payload editors, and metadata fields.

Use it as `{% jsoninput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'jsoninput', …)`.

Basic field

A label and a placeholder. With `formatOnBlur` on, valid JSON is pretty-printed when the field loses focus; `autosize` plus `minRows` sizes the box.

Config

Type some JSON and click away — valid JSON is reformatted; invalid JSON shows the validation message.

Source: Markdown

```
{% jsoninput label='Config' placeholder='{ "key": "value" }' formatOnBlur=true autosize=true
minRows=4 %}
```

Source: Python

```
component('aardvark', 'jsoninput', label='Config',
          placeholder='{ "key": "value" }', formatOnBlur=True,
          autosize=True, minRows=4)
```

Custom validation message

`validationError` is the message shown when the content isn't valid JSON.

Payload

Source: Markdown

```
{% jsoninput label='Payload' validationError='That is not valid JSON' formatOnBlur=true
minRows=3 defaultValue='{ broken: }' %}
```

Source: Python

```
component('aardvark', 'jsoninput', label='Payload',
          validationError='That is not valid JSON', formatOnBlur=True,
          minRows=3, defaultValue='{ broken: }')
```

Autosize bounds

`minRows` sets the floor and `maxRows` the cap when `autosize` grows the field with its content.

Metadata

Source: Markdown

```
{% jsoninput label='Metadata' autosize=true minRows=2 maxRows=10 placeholder='{ }'
formatOnBlur=true %}
```

Source: Python

```
component('aardvark', 'jsoninput', label='Metadata', autosize=True,
          minRows=2, maxRows=10, placeholder='{ }', formatOnBlur=True)
```

Required, error, and disabled

`required` and `withAsterisk` add the asterisk; `error` shows a wrapper-level validation message; `disabled` greys the control out.

Webhook body

Rules

Frozen config

Source: Markdown

```
{% jsoninput label='Webhook body' required=true minRows=3 placeholder='{ }' %}

{% jsoninput label='Rules' error='At least one rule is required' minRows=3 defaultValue='[]'
%}

{% jsoninput label='Frozen config' disabled=true minRows=3 defaultValue='{ "env": "prod" }'
%}
```

Source: Python

```
component('aardvark', 'jsoninput', label='Webhook body', required=True,
          minRows=3, placeholder='{ }')

component('aardvark', 'jsoninput', label='Rules',
          error='At least one rule is required', minRows=3, defaultValue='[]')

component('aardvark', 'jsoninput', label='Frozen config', disabled=True,
          minRows=3, defaultValue='{ "env": "prod" }')
```

Variants, size, and radius

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`.

Filled

Large, round

Source: Markdown

```
{% jsoninput label='Filled' variant='filled' minRows=3 placeholder='{ }' %}

{% jsoninput label='Large, round' size='lg' radius='lg' minRows=3 placeholder='{ }' %}
```

Source: Python

```
component('aardvark', 'jsoninput', label='Filled', variant='filled',
          minRows=3, placeholder='{ }')

component('aardvark', 'jsoninput', label='Large, round', size='lg',
          radius='lg', minRows=3, placeholder='{ }')
```

With other components

Pair a name field with a JSON config editor inside a {% card %}.

New integration

Name

Settings (JSON)

Source: Markdown

```
{% card title='New integration' %}
{% textinput label='Name' placeholder='My integration' required=true %}

{% jsoninput label='Settings (JSON)' formatOnBlur=true autosize=true minRows=4
placeholder='{ "enabled": true }' %}
{% endCard %}
```

Source: Python

```
name = component('aardvark', 'textinput', label='Name',
                 placeholder='My integration', required=True)
settings = component('aardvark', 'jsoninput', label='Settings (JSON)',
                     formatOnBlur=True, autosize=True, minRows=4,
```

```
                    placeholder='{ "enabled": true }')
component('aardvark', 'card', title='New integration',
          children=name + settings)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when the field is empty.
`error`	string	Wrapper-level validation message; switches the field to the error color.
`defaultValue`	string	Initial value of the field.
`formatOnBlur`	bool (`true` / `false`)	Pretty-print valid JSON when the field loses focus.
`validationError`	string	Message shown when the content isn't valid JSON.
`autosize`	bool (`true` / `false`)	Grow the field with its content.
`minRows`	integer	Minimum number of visible rows (a floor when `autosize` is on).
`maxRows`	integer	Maximum number of rows the field grows to (with `autosize`).
`variant`	`default`, `filled`, `unstyled`	Visual style of the input.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS length	Corner radius.
`required`	bool (`true` / `false`)	Mark the field required and add the asterisk.

`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required` attribute.
`disabled`	bool (`true` / `false`)	Render the field disabled.

CSS Selectors

Target a `{% jsoninput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every JsonInput instance on the page */
[data-aardvark-island="JsonInput"] { }

/* Mantine Styles API parts */
.mantine-JsonInput-root { }
.mantine-JsonInput-input { }
.mantine-JsonInput-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Config

Source: Markdown

```
{% jsoninput label='Config' placeholder='{ "key": "value" }' formatOnBlur=true autosize=true
minRows=4 attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'jsoninput', label='Config',
          placeholder='{ "key": "value" }', formatOnBlur=True,
          autosize=True, minRows=4, attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

MaskInput

A masked text field — entry is constrained to a fixed pattern as the reader types, with separators inserted automatically. Reach for it for phone numbers, dates, and card numbers. It carries the same Input wrapper as TextInput — label, description, error, sections — plus the `mask` pattern.

Use it as `{% maskinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'maskinput', …)`.

Mask tokens

`mask` is a pattern where each token constrains one character and every other character is a literal separator inserted for you.

Token	Matches
`9`	A digit (`0`–`9`).
`a`	A letter (`A`–`Z`, `a`–`z`).
`A`	An uppercase letter (`A`–`Z`).
`*`	A letter or a digit.
`#`	A digit or a sign (`0`–`9`, `+`, `-`).
anything else	A literal separator, inserted automatically.

Basic field

A phone mask: type digits and the parentheses, space, and dash appear on their own.

Phone

Source: Markdown

```
{% maskinput label='Phone' mask='(999) 999-9999' placeholder='(___) ___-____' %}
```

Source: Python

```
component('aardvark', 'maskinput', label='Phone',
          mask='(999) 999-9999', placeholder='(___) ___-____')
```

More patterns

A date mask uses only `9`; a card mask groups digits; a license mask mixes letters and digits.

Date

Card

License

Serial

Source: Markdown

```
{% maskinput label='Date' mask='99/99/9999' placeholder='MM/DD/YYYY' %}

{% maskinput label='Card' mask='9999 9999 9999 9999' placeholder='0000 0000 0000 0000' %}

{% maskinput label='License' mask='aaa-9999' description='Three letters, a dash, four
digits' %}

{% maskinput label='Serial' mask='****-****' description='Letters or digits' %}
```

Source: Python

```
component('aardvark', 'maskinput', label='Date',
          mask='99/99/9999', placeholder='MM/DD/YYYY')

component('aardvark', 'maskinput', label='Card',
          mask='9999 9999 9999 9999', placeholder='0000 0000 0000 0000')

component('aardvark', 'maskinput', label='License', mask='aaa-9999',
          description='Three letters, a dash, four digits')

component('aardvark', 'maskinput', label='Serial', mask='****-****',
          description='Letters or digits')
```

Default value

`defaultValue` seeds the field; the value is masked on render.

Phone

Source: Markdown

```
{% maskinput label='Phone' mask='(999) 999-9999' defaultValue='(555) 010-0199' %}
```

Source: Python

```
component('aardvark', 'maskinput', label='Phone',
          mask='(999) 999-9999', defaultValue='(555) 010-0199')
```

Required, error, and disabled

`required` and `withAsterisk` add the asterisk; `error` shows a validation message; `disabled` greys the control out.

Phone

Card

On-file number

Source: Markdown

```
{% maskinput label='Phone' mask='(999) 999-9999' required=true placeholder='(___) ___-____'
%}

{% maskinput label='Card' mask='9999 9999 9999 9999' error='Card number is incomplete'
placeholder='0000 0000 0000 0000' %}

{% maskinput label='On-file number' mask='(999) 999-9999' disabled=true defaultValue='(555)
010-0199' %}
```

Source: Python

```
component('aardvark', 'maskinput', label='Phone', mask='(999) 999-9999',
          required=True, placeholder='(___) ___-____')

component('aardvark', 'maskinput', label='Card', mask='9999 9999 9999 9999',
          error='Card number is incomplete',
          placeholder='0000 0000 0000 0000')

component('aardvark', 'maskinput', label='On-file number',
          mask='(999) 999-9999', disabled=True, defaultValue='(555) 010-0199')
```

Variants, size, radius, and sections

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`; `leftSection`/`rightSection` add text inside the field.

Filled

Large, round

Phone

Source: Markdown

```
{% maskinput label='Filled' mask='99/99/9999' variant='filled' placeholder='MM/DD/YYYY' %}
```

```
{% maskinput label='Large, round' mask='(999) 999-9999' size='lg' radius='xl'
placeholder='(___) ___-____' %}

{% maskinput label='Phone' mask='999 999-9999' leftSection='+1' placeholder='000 000-0000'
%}
```

Source: Python

```
component('aardvark', 'maskinput', label='Filled', mask='99/99/9999',
          variant='filled', placeholder='MM/DD/YYYY')

component('aardvark', 'maskinput', label='Large, round',
          mask='(999) 999-9999', size='lg', radius='xl',
          placeholder='(___) ___-____')

component('aardvark', 'maskinput', label='Phone', mask='999 999-9999',
          leftSection='+1', placeholder='000 000-0000')
```

With other components

Combine a masked phone field with a TextInput inside a {% card %}.

Contact

Full name

Phone

Source: Markdown

```
{% card title='Contact' %}
{% textinput label='Full name' placeholder='Ada Lovelace' required=true %}

{% maskinput label='Phone' mask='(999) 999-9999' placeholder='(___) ___-____' required=true
%}
{% endCard %}
```

Source: Python

```
name = component('aardvark', 'textinput', label='Full name',
                 placeholder='Ada Lovelace', required=True)
phone = component('aardvark', 'maskinput', label='Phone',
                  mask='(999) 999-9999', placeholder='(___) ___-____',
                  required=True)
component('aardvark', 'card', title='Contact', children=name + phone)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`mask`	string of tokens (`9`, `a`, `A`, `*`, `#`) plus literal separators	The entry pattern (see the tokens table above).
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when the field is empty.
`error`	string	Validation message below the field; switches it to the error color.
`defaultValue`	string	Initial value (masked on render).
`variant`	`default`, `filled`, `unstyled`	Visual style of the input.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS length	Corner radius.
`required`	bool (`true` / `false`)	Mark the field required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required` attribute.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`leftSection`	string	Text shown inside the field, before the value.
`rightSection`	string	Text shown inside the field, after the value.

CSS Selectors

Target a `{% maskinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every MaskInput instance on the page */
[data-aardvark-island="MaskInput"] { }

/* Mantine Styles API parts */
.mantine-MaskInput-root { }
.mantine-MaskInput-input { }
.mantine-MaskInput-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Phone

Source: Markdown

```
{% maskinput label='Phone' mask='(999) 999-9999' placeholder='(___) ___-____'
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'maskinput', label='Phone',
          mask='(999) 999-9999', placeholder='(___) ___-____', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

NativeSelect

A built-in tag for a native HTML `<select>` styled to match the rest of your form. It uses the same Input wrapper as TextInput, so it carries a label, description, and error message, and builds its options from a `data` string. Because it renders the browser's native control, it is the lightest, most accessible way to offer a fixed list of choices.

Use it as `{% nativeselect %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'nativeselect', …)`.

Options from data

The options come from `data` — a `|`- or comma-delimited string. Each option is trimmed.

Framework

Source: Markdown

```
{% nativeselect label='Framework' data='React|Vue|Svelte|Angular' %}
```

Source: Python

```
component('aardvark', 'nativeselect', label='Framework', data='React|Vue|Svelte|Angular')
```

Delimiters

Comma-delimited works too. When you use `|`, an option label may itself contain a comma.

Size

City

Source: Markdown

```
{% nativeselect label='Size' data='Small, Medium, Large' description='Comma-delimited' %}

{% nativeselect label='City' data='Paris, France|Rome, Italy|Tokyo, Japan'
description='Pipe-delimited so labels can hold commas' %}
```

Source: Python

```
component('aardvark', 'nativeselect', label='Size', data='Small, Medium, Large',
description='Comma-delimited')

component(
    'aardvark', 'nativeselect',
    label='City',
    data='Paris, France|Rome, Italy|Tokyo, Japan',
```

```
    description='Pipe-delimited so labels can hold commas',
)
```

Pre-select and state

`defaultValue` pre-selects an option; `required` (or `withAsterisk`) adds the asterisk; `error` shows a validation message and switches the field to the error color.

Tier

Plan

Source: Markdown

```
{% nativeselect label='Tier' data='Free|Pro|Enterprise' defaultValue='Pro' %}

{% nativeselect label='Plan' data='Free|Pro|Enterprise' required=true error='Choose a plan
to continue' %}
```

Source: Python

```
component('aardvark', 'nativeselect', label='Tier', data='Free|Pro|Enterprise',
defaultValue='Pro')

component(
    'aardvark', 'nativeselect',
    label='Plan',
    data='Free|Pro|Enterprise',
    required=True,
    error='Choose a plan to continue',
)
```

Variant, size, radius, and sections

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`; `leftSection` and `rightSection` put text inside the field; `disabled` renders it inert.

Filled

Currency

Locked

Source: Markdown

```
{% nativeselect label='Filled' data='One|Two|Three' variant='filled' %}

{% nativeselect label='Currency' data='USD|EUR|GBP' leftSection='' size='lg' radius='md' %}
```

```
{% nativeselect label='Locked' data='Free|Pro' disabled=true %}
```

Source: Python

```
component('aardvark', 'nativeselect', label='Filled', data='One|Two|Three',
variant='filled')

component('aardvark', 'nativeselect', label='Currency', data='USD|EUR|GBP', leftSection='',
size='lg', radius='md')

component('aardvark', 'nativeselect', label='Locked', data='Free|Pro', disabled=True)
```

With other components

Group a select with other fields inside a fieldset.

Theme

Language

Source: Markdown

```
{% fieldset legend='Preferences' %}
{% nativeselect label='Theme' data='System|Light|Dark' defaultValue='System' %}
{% nativeselect label='Language' data='English|Français|' %}
{% endFieldset %}
```

Source: Python

```
inner = (
    component('aardvark', 'nativeselect', label='Theme', data='System|Light|Dark',
defaultValue='System')
    + component('aardvark', 'nativeselect', label='Language', data='English|Français|')
)
component('aardvark', 'fieldset', legend='Preferences', children=inner)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`data`	string	Options as a `|`- or comma-delimited string. Use `|` when an option label contains a comma.
`defaultValue`	string	The option pre-selected on load.

`label`	string	Label shown above the field.
`description`	string	Help text shown under the label.
`error`	string	Validation message shown below the field; switches it to the error color.
`variant`	`default`, `filled`, `unstyled`	Visual style.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Field size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl` (or any CSS value)	Corner radius.
`required`	`true` / `false`	Mark the field required and add the asterisk. Default `false`.
`withAsterisk`	`true` / `false`	Add the asterisk without the `required` semantics. Default `false`.
`disabled`	`true` / `false`	Render the field disabled. Default `false`.
`leftSection`	string	Text shown inside the field on the left.
`rightSection`	string	Text shown inside the field on the right.

CSS Selectors

Target a `{% nativeselect %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every NativeSelect instance on the page */
[data-aardvark-island="NativeSelect"] { }

/* Mantine Styles API parts */
.mantine-NativeSelect-root { }
.mantine-NativeSelect-input { }
.mantine-NativeSelect-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Framework

Source: Markdown

```
{% nativeselect label='Framework' data='React|Vue|Svelte|Angular' attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'nativeselect', label='Framework', data='React|Vue|Svelte|Angular',
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

NumberInput

A numeric field with increment/decrement controls, bounds, and formatting. It carries the same Input wrapper as TextInput and adds the numeric specifics: `min`/`max`/`step`, `decimalScale`, `prefix`/`suffix`, `thousandSeparator`, `allowNegative`/`allowDecimal`, `clampBehavior`, and `hideControls`. Reach for it for quantities, prices, ratings, and percentages.

Use it as `{% numberinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'numberinput', …)`.

Basic field

A label, bounds, a step, and a starting value.

Quantity

Source: Markdown

```
{% numberinput label='Quantity' min=0 max=100 step=1 defaultValue=1 %}
```

Source: Python

```
component('aardvark', 'numberinput', label='Quantity',
          min=0, max=100, step=1, defaultValue=1)
```

Bounds, step, and precision

`min`, `max`, and `step` constrain the value; `decimalScale` fixes the number of decimal places; `clampBehavior` is `strict` (keep the value inside the bounds while typing), `blur` (clamp on blur), or `none`.

Rating (0–5, halves)

Strict clamp

Source: Markdown

```
{% numberinput label='Rating (0-5, halves)' min=0 max=5 step=0.5 decimalScale=1
defaultValue=2.5 %}

{% numberinput label='Strict clamp' min=1 max=10 clampBehavior='strict' defaultValue=5 %}
```

Source: Python

```
component('aardvark', 'numberinput', label='Rating (0-5, halves)',
          min=0, max=5, step=0.5, decimalScale=1, defaultValue=2.5)

component('aardvark', 'numberinput', label='Strict clamp',
```

```
          min=1, max=10, clampBehavior='strict', defaultValue=5)
```

Prefix, suffix, and separators

`prefix` and `suffix` wrap the value; `thousandSeparator` groups digits.

Price

Weight

Source: Markdown

```
{% numberinput label='Price' prefix='$' thousandSeparator=',' decimalScale=2
defaultValue=1999.5 %}

{% numberinput label='Weight' suffix=' kg' min=0 defaultValue=72 %}
```

Source: Python

```
component('aardvark', 'numberinput', label='Price', prefix='$',
          thousandSeparator=',', decimalScale=2, defaultValue=1999.5)

component('aardvark', 'numberinput', label='Weight', suffix=' kg',
          min=0, defaultValue=72)
```

Sign and decimals

`allowNegative=false` blocks negatives; `allowDecimal=false` blocks the decimal point (whole numbers only).

Whole, positive only

Temperature (°C)

Source: Markdown

```
{% numberinput label='Whole, positive only' allowNegative=false allowDecimal=false
defaultValue=5 %}

{% numberinput label='Temperature (°C)' allowNegative=true step=0.1 decimalScale=1
defaultValue=-4.5 %}
```

Source: Python

```
component('aardvark', 'numberinput', label='Whole, positive only',
          allowNegative=False, allowDecimal=False, defaultValue=5)

component('aardvark', 'numberinput', label='Temperature (°C)',
          allowNegative=True, step=0.1, decimalScale=1, defaultValue=-4.5)
```

Hide the stepper controls

`hideControls` removes the increment/decrement buttons, leaving a plain numeric field.

No controls

Source: Markdown

```
{% numberinput label='No controls' hideControls=true defaultValue=10 %}
```

Source: Python

```
component('aardvark', 'numberinput', label='No controls',
          hideControls=True, defaultValue=10)
```

Required, error, and disabled

`required` and `withAsterisk` add the asterisk; `error` shows a validation message; `disabled` greys the control out.

Seats

Discount %

Plan limit

Source: Markdown

```
{% numberinput label='Seats' required=true min=1 defaultValue=1 %}

{% numberinput label='Discount %' error='Must be 100 or less' max=100 defaultValue=120 %}

{% numberinput label='Plan limit' disabled=true defaultValue=50 %}
```

Source: Python

```
component('aardvark', 'numberinput', label='Seats', required=True,
          min=1, defaultValue=1)

component('aardvark', 'numberinput', label='Discount %',
          error='Must be 100 or less', max=100, defaultValue=120)

component('aardvark', 'numberinput', label='Plan limit',
          disabled=True, defaultValue=50)
```

Variants, size, radius, and sections

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`; `leftSection`/`rightSection` add text inside the field.

Filled

Large, round

CPU cores

Source: Markdown

```
{% numberinput label='Filled' variant='filled' defaultValue=42 %}

{% numberinput label='Large, round' size='lg' radius='xl' defaultValue=7 %}

{% numberinput label='CPU cores' leftSection='⎚' rightSection='cores' min=1 defaultValue=4
%}
```

Source: Python

```
component('aardvark', 'numberinput', label='Filled', variant='filled',
          defaultValue=42)

component('aardvark', 'numberinput', label='Large, round', size='lg',
          radius='xl', defaultValue=7)

component('aardvark', 'numberinput', label='CPU cores', leftSection='⎚',
          rightSection='cores', min=1, defaultValue=4)
```

With other components

Combine a price field with a quantity field inside a {% card %}.

Order line

Unit price

Quantity

Source: Markdown

```
{% card title='Order line' %}
{% numberinput label='Unit price' prefix='$' decimalScale=2 min=0 defaultValue=19.99 %}

{% numberinput label='Quantity' min=1 max=99 defaultValue=1 %}
```

```
{% endCard %}
```

Source: Python

```
price = component('aardvark', 'numberinput', label='Unit price', prefix='$',
                  decimalScale=2, min=0, defaultValue=19.99)
qty = component('aardvark', 'numberinput', label='Quantity',
                min=1, max=99, defaultValue=1)
component('aardvark', 'card', title='Order line', children=price + qty)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when the field is empty.
`error`	string	Validation message below the field; switches it to the error color.
`defaultValue`	number	Initial numeric value.
`min`	number	Lowest allowed value.
`max`	number	Highest allowed value.
`step`	number	Increment applied by the stepper buttons and arrow keys.
`decimalScale`	integer	Fixed number of decimal places to display.
`prefix`	string	Text shown immediately before the value.
`suffix`	string	Text shown immediately after the value.
`thousandSeparator`	string (e.g. `,`)	Digit-group separator.
`allowNegative`	bool (`true` / `false`)	Allow negative values (default `true`; set `false` to block).

`allowDecimal`	bool (`true` / `false`)	Allow a decimal point (default `true`; set `false` for whole numbers).
`hideControls`	bool (`true` / `false`)	Remove the increment/decrement stepper buttons.
`clampBehavior`	`strict`, `blur`, `none`	When the value is clamped to `min`/`max`.
`variant`	`default`, `filled`, `unstyled`	Visual style of the input.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS length	Corner radius.
`required`	bool (`true` / `false`)	Mark the field required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required` attribute.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`leftSection`	string	Text shown inside the field, before the value.
`rightSection`	string	Text shown inside the field, after the value.

CSS Selectors

Target a `{% numberinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every NumberInput instance on the page */
[data-aardvark-island="NumberInput"] { }

/* Mantine Styles API parts */
.mantine-NumberInput-root { }
.mantine-NumberInput-input { }
.mantine-NumberInput-control { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Quantity

Source: Markdown

```
{% numberinput label='Quantity' min=0 max=100 step=1 defaultValue=1 attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'numberinput', label='Quantity',
          min=0, max=100, step=1, defaultValue=1, attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

PasswordInput

A password field with a built-in show/hide visibility toggle on the right. It carries the same Input wrapper as TextInput — label, description, error, `leftSection` — plus `defaultVisible` to start the field revealed. The toggle button occupies the right section, so only `leftSection` is exposed.

Use it as `{% passwordinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'passwordinput', …)`.

Basic field

A label and a placeholder; click the eye icon to reveal the value.

Password

Source: Markdown

```
{% passwordinput label='Password' placeholder='Your password' %}
```

Source: Python

```
component('aardvark', 'passwordinput', label='Password',
          placeholder='Your password')
```

Start revealed

`defaultVisible` opens the field with the value shown — handy for an API key or token the reader needs to copy.

API key

Source: Markdown

```
{% passwordinput label='API key' defaultVisible=true defaultValue='sk-demo-1234' %}
```

Source: Python

```
component('aardvark', 'passwordinput', label='API key',
          defaultVisible=True, defaultValue='sk-demo-1234')
```

Required, asterisk, and error

`required` and `withAsterisk` add the asterisk; `error` shows a validation message and switches the field to the error color.

New password

Confirm password

New password

Source: Markdown

```
{% passwordinput label='New password' required=true placeholder='8+ characters' %}

{% passwordinput label='Confirm password' withAsterisk=true placeholder='Repeat it' %}

{% passwordinput label='New password' error='Too weak — add a number' placeholder='8+
characters' %}
```

Source: Python

```
component('aardvark', 'passwordinput', label='New password', required=True,
          placeholder='8+ characters')

component('aardvark', 'passwordinput', label='Confirm password',
          withAsterisk=True, placeholder='Repeat it')

component('aardvark', 'passwordinput', label='New password',
          error='Too weak — add a number', placeholder='8+ characters')
```

Disabled

`disabled` greys the control out.

Vault password

Source: Markdown

```
{% passwordinput label='Vault password' disabled=true defaultValue='locked' %}
```

Source: Python

```
component('aardvark', 'passwordinput', label='Vault password',
          disabled=True, defaultValue='locked')
```

Variants, size, radius, and section

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`; `leftSection` adds text before the value (the right side holds the toggle).

Filled

Large, round

PIN

Source: Markdown

```
{% passwordinput label='Filled' variant='filled' placeholder='filled variant' %}

{% passwordinput label='Large, round' size='lg' radius='xl' placeholder='lg / xl radius' %}

{% passwordinput label='PIN' leftSection='' placeholder='Enter PIN' %}
```

Source: Python

```
component('aardvark', 'passwordinput', label='Filled', variant='filled',
          placeholder='filled variant')

component('aardvark', 'passwordinput', label='Large, round', size='lg',
          radius='xl', placeholder='lg / xl radius')

component('aardvark', 'passwordinput', label='PIN', leftSection='',
          placeholder='Enter PIN')
```

With other components

Build a sign-in form with a TextInput and a password field inside a {% card %}.

Sign in

Email

Password

Source: Markdown

```
{% card title='Sign in' %}
{% textinput label='Email' type='email' placeholder='you@example.com' required=true %}

{% passwordinput label='Password' placeholder='Your password' required=true %}
{% endCard %}
```

Source: Python

```
email = component('aardvark', 'textinput', label='Email', type='email',
                  placeholder='you@example.com', required=True)
pw = component('aardvark', 'passwordinput', label='Password',
               placeholder='Your password', required=True)
component('aardvark', 'card', title='Sign in', children=email + pw)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when the field is empty.
`error`	string	Validation message below the field; switches it to the error color.
`defaultValue`	string	Initial value of the field.
`defaultVisible`	bool (`true` / `false`)	Start with the value revealed instead of masked.
`variant`	`default`, `filled`, `unstyled`	Visual style of the input.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS length	Corner radius.
`required`	bool (`true` / `false`)	Mark the field required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required` attribute.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`leftSection`	string	Text shown inside the field, before the value (the right side holds the visibility toggle).

CSS Selectors

Target a `{% passwordinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every PasswordInput instance on the page */
[data-aardvark-island="PasswordInput"] { }
```

```
/* Mantine Styles API parts */
.mantine-PasswordInput-root { }
.mantine-PasswordInput-innerInput { }
.mantine-PasswordInput-visibilityToggle { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it. For a password field that's a demonstration only — never log or transmit a real password value in production:

Password

Source: Markdown

```
{% passwordinput label='Password' placeholder='Your password' attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'passwordinput', label='Password',
          placeholder='Your password', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

PinInput

A built-in tag for one-time-code or PIN entry — a row of single-character boxes that advance as you type. It is not an Input wrapper, so it has no label or message of its own; its `error` is a boolean that adds error styling to every box. Pair it with a heading or a textinput label when you need a caption.

Use it as `{% pininput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'pininput', …)`.

Basic PIN

`length` sets the number of boxes (four by default).

Source: Markdown

```
{% pininput length=4 %}
```

Source: Python

```
component('aardvark', 'pininput', length=4)
```

Length and type

`type` is `number` (default, digits only) or `alphanumeric` (letters and digits).

Source: Markdown

```
{% pininput length=6 type='number' %}

{% pininput length=5 type='alphanumeric' %}
```

Source: Python

```
component('aardvark', 'pininput', length=6, type='number')

component('aardvark', 'pininput', length=5, type='alphanumeric')
```

Mask and OTP autofill

`mask` obscures entered characters like a password; `oneTimeCode` opts into the browser's one-time-code autofill so an SMS code can be filled automatically.

Source: Markdown

```
{% pininput length=4 mask=true oneTimeCode=true %}
```

Source: Python

```
component('aardvark', 'pininput', length=4, mask=True, oneTimeCode=True)
```

Placeholder, size, radius, and gap

`placeholder` sets the per-box placeholder character; `size` and `radius` style each box; `gap` is the space between boxes.

Source: Markdown

```
{% pininput length=4 placeholder='○' size='lg' radius='xl' gap='md' %}
```

Source: Python

```
component('aardvark', 'pininput', length=4, placeholder='○', size='lg', radius='xl',
gap='md')
```

Error and disabled

`error` is a boolean that adds error styling and `aria-invalid` to every box; `disabled` makes the whole row inert.

Source: Markdown

```
{% pininput length=4 error=true %}

{% pininput length=4 disabled=true %}
```

Source: Python

```
component('aardvark', 'pininput', length=4, error=True)

component('aardvark', 'pininput', length=4, disabled=True)
```

With other components

PinInput has no label of its own, so pair it with a heading and supporting text inside a card to make a complete verification prompt.

Verify your email

We sent a 6-digit code to your inbox.

Source: Markdown

```
{% card title='Verify your email' %}
We sent a 6-digit code to your inbox.

{% pininput length=6 type='number' oneTimeCode=true %}
{% endCard %}
```

Source: Python

```
body = (
    'We sent a 6-digit code to your inbox.\n\n'
    + component('aardvark', 'pininput', length=6, type='number', oneTimeCode=True)
)
component('aardvark', 'card', title='Verify your email', children=body)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`length`	integer	Number of single-character boxes. Default `4`.
`type`	`number`, `alphanumeric`	Accepted characters. `number` (default) is digits only; `alphanumeric` allows letters and digits.
`mask`	`true` / `false`	Obscure entered characters like a password. Default `false`.
`oneTimeCode`	`true` / `false`	Opt into the browser's one-time-code (OTP) autofill. Default `false`.
`placeholder`	string	Per-box placeholder character.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Box size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl` (or any CSS value)	Box corner radius.

`gap`	`xs`, `sm`, `md`, `lg`, `xl` (or any CSS value)	Space between boxes.
`disabled`	`true` / `false`	Disable every box. Default `false`.
`error`	`true` / `false`	Add error styling and `aria-invalid` to every box. Boolean — there is no wrapper for a message. Default `false`.

CSS Selectors

Target a `{% pininput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every PinInput instance on the page */
[data-aardvark-island="PinInput"] { }

/* Mantine Styles API parts */
.mantine-PinInput-root { }
.mantine-PinInput-pinInput { }
.mantine-PinInput-input { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it. For a PIN field that's a demonstration only — never log or transmit a real PIN value in production:

Source: Markdown

```
{% pininput length=4 attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'pininput', length=4, attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

Radio

A single radio button with a label. The label is the `label` attribute or the block body. Give several radios the **same** `name` to make them a mutually exclusive group, the way standard HTML radios behave. Add `defaultChecked` to pick the option selected on load.

Use it as `{% radio %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'radio', …)`.

A radio group

Give each radio the same `name` so selecting one deselects the others. `value` is what that option submits; `defaultChecked` picks the starting selection.

Email

Phone

Mail

Source: Markdown

```
{% radio label='Email' name='contact' value='email' defaultChecked=true %}
{% radio label='Phone' name='contact' value='phone' %}
{% radio label='Mail' name='contact' value='mail' %}
```

Source: Python

```
component('aardvark', 'radio', label='Email', name='contact', value='email',
          defaultChecked=True)
component('aardvark', 'radio', label='Phone', name='contact', value='phone')
component('aardvark', 'radio', label='Mail', name='contact', value='mail')
```

In a loop, build the group from data and select the first option:

Source: Python

```
for i, opt in enumerate(['Email', 'Phone', 'Mail']):
    component('aardvark', 'radio', label=opt, name='contact', value=opt.lower(),
              defaultChecked=(i == 0))
```

Colors and sizes

`color` is any theme color for the selected dot; `size` is `xs`–`xl`. (Each example uses a distinct `name` so they don't fight over one selection.)

blue

grape

large

Source: Markdown

```
{% radio label='blue' name='demo' defaultChecked=true %}
{% radio label='grape' name='demo2' color='grape' defaultChecked=true %}
{% radio label='large' name='demo3' size='lg' defaultChecked=true %}
```

Source: Python

```
component('aardvark', 'radio', label='blue', name='demo', defaultChecked=True)
component('aardvark', 'radio', label='grape', name='demo2', color='grape',
          defaultChecked=True)
component('aardvark', 'radio', label='large', name='demo3', size='lg',
          defaultChecked=True)
```

Label position, icon color, helper text, and disabled

`labelPosition='left'` puts the label first; `iconColor` colors the inner dot; `description` and `error` add helper text; `disabled` makes it non-interactive.

Label on the left

Custom dot color

With a description

Disabled

Source: Markdown

```
{% radio label='Label on the left' name='lp' labelPosition='left' defaultChecked=true %}
{% radio label='Custom dot color' name='ic' iconColor='yellow' color='dark'
defaultChecked=true %}
{% radio label='With a description' name='ds' description='We will only call about your
order.' defaultChecked=true %}
{% radio label='Disabled' name='db' disabled=true defaultChecked=true %}
```

Source: Python

```
component('aardvark', 'radio', label='Label on the left', name='lp',
          labelPosition='left', defaultChecked=True)
component('aardvark', 'radio', label='Custom dot color', name='ic',
          iconColor='yellow', color='dark', defaultChecked=True)
component('aardvark', 'radio', label='With a description', name='ds',
          description='We will only call about your order.', defaultChecked=True)
component('aardvark', 'radio', label='Disabled', name='db', disabled=True,
          defaultChecked=True)
```

With other components

Combine a radio group with a heading and a button to build a small choice form.

How should we reach you?

Email

Phone

Save preference

Source: Markdown

```
How should we reach you?

{% radio label='Email' name='reach' value='email' defaultChecked=true %}
{% radio label='Phone' name='reach' value='phone' %}

{% button %}Save preference{% endButton %}
```

Source: Python

```
component('aardvark', 'radio', label='Email', name='reach', value='email',
          defaultChecked=True)
component('aardvark', 'radio', label='Phone', name='reach', value='phone')
component('aardvark', 'button', children='Save preference')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	The label, when not using the block body. Plain text — Markdown (e.g. `**bold**`) is not rendered.
`name`	string	Radio-group name — give grouped radios the same value to make them mutually exclusive.
`value`	string	The value this option submits.
`defaultChecked`	bare flag (`true`)	Start this option selected.
`disabled`	bare flag (`true`)	Render non-interactive.

`color`	theme color name (`blue`, `grape`, ...)	Fill color of the selected dot.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Overall size.
`labelPosition`	`right` (default), `left`	Which side of the button the label sits on.
`iconColor`	theme color name	Color of the inner dot.
`description`	string	Helper text shown below the label.
`error`	string	Error text shown below the label (also marks the option invalid).
`autoContrast`	bare flag (`true`)	Auto-pick a readable dot color against `color`.
`attr`	dict (`attr={…}`)	Raw HTML attributes (e.g. `onchange`) applied to the rendered root.

CSS Selectors

Target a `{% radio %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Radio instance on the page */
[data-aardvark-island="Radio"] { }

/* Mantine Styles API parts */
.mantine-Radio-root { }
.mantine-Radio-radio { }
.mantine-Radio-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so selecting an option logs its value to the console and alerts it:

Email

SMS

Source: Markdown

```
{% radio label='Email' name='contact' value='email' attr={'onchange': '''
```

```
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% radio label='SMS' name='contact' value='sms' attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
print(component('aardvark', 'radio', label='Email', name='contact', value='email',
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''}))
print(component('aardvark', 'radio', label='SMS', name='contact', value='sms',
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''}))
```

RangeSlider

A **built-in** tag for a range slider — let the reader pick a low and high bound by dragging either of two thumbs. It ships with Aardvark, so it's a single tag with no setup. The `defaultValue` is a pair of numbers (the starting low and high), and `minRange` keeps the two thumbs a minimum distance apart.

Use it as `{% rangeslider %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'rangeslider', …)`.

Demonstrations

Basic range

Set `min`, `max`, and a `defaultValue` of **two numbers** — the starting low and high. Drag either thumb.

Source: Markdown

```
{% rangeslider min=0 max=100 defaultValue='20, 80' %}
```

Source: Python

The `defaultValue` pair stays a `'low, high'` string — the macro parses it into a two-number list:

```
component('aardvark', 'rangeslider', min=0, max=100, defaultValue='20, 80')
```

Marks and a minimum gap

Pass `marks` as a JSON array of `{value, label}`, and `minRange` to fix the smallest gap the two thumbs may have. Add `restrictToMarks` to snap to the marks only.

Source: Markdown

```
{% rangeslider min=0 max=100 defaultValue='25, 75' minRange=10 color='teal'
marks='[{"value": 0, "label": "0"}, {"value": 50, "label": "50"}, {"value": 100, "label":
"100"}]' %}
```

Source: Python

```
component('aardvark', 'rangeslider',
          min=0, max=100, defaultValue='25, 75', minRange=10, color='teal',
          marks='[{"value": 0, "label": "0"}, {"value": 50, "label": "50"}, {"value": 100,
"label": "100"}]')
```

Colors, sizes, and inverted fill

`color` paints the filled span between the thumbs; `size` and `radius` take `xs`–`xl`. Add `inverted` to fill the outside instead of the inside, and `labelAlwaysOn` to keep both value bubbles visible.

Source: Markdown

```
{% rangeslider defaultValue='30, 60' size='lg' color='grape' labelAlwaysOn=true %}
{% rangeslider defaultValue='30, 70' color='orange' inverted=true %}
```

Source: Python

```
component('aardvark', 'rangeslider', defaultValue='30, 60', size='lg', color='grape',
labelAlwaysOn=True)
component('aardvark', 'rangeslider', defaultValue='30, 70', color='orange', inverted=True)
```

Disabled

Add the bare `disabled` flag to make the range read-only.

Source: Markdown

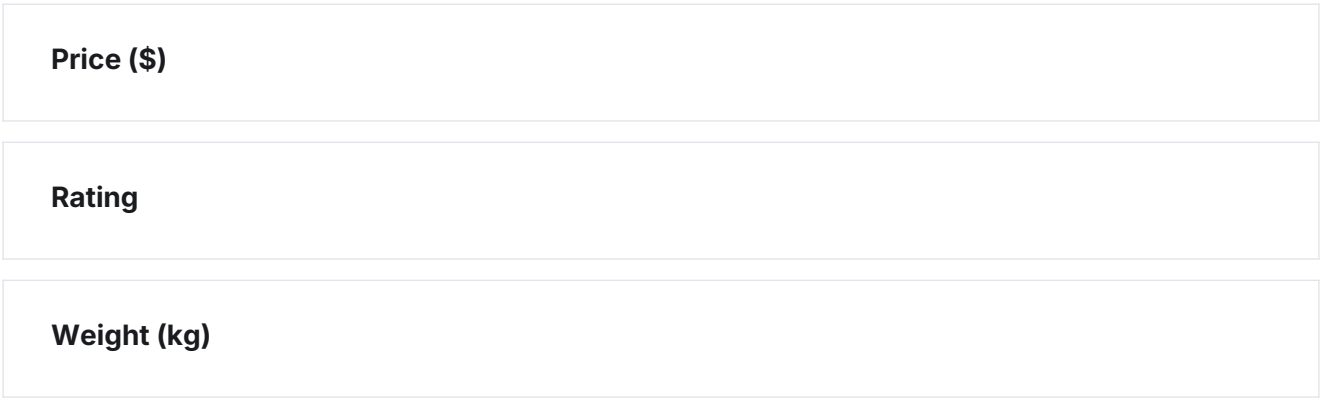
```
{% rangeslider defaultValue='40, 60' disabled=true %}
```

Source: Python

```
component('aardvark', 'rangeslider', defaultValue='40, 60', disabled=True)
```

With other components

Generate one range slider per filter from data with a Python loop — the same `component('aardvark', 'rangeslider', …)` call, once per item, laid out in a card grid.



Source: Python

```
filters = [
```

```
    {"name": "Price ($)", "min": 0, "max": 500, "default": "100, 350", "color": "blue"},
    {"name": "Rating", "min": 0, "max": 5, "default": "3, 5", "color": "teal"},
    {"name": "Weight (kg)", "min": 0, "max": 20, "default": "2, 8", "color": "grape"},
]
controls = ''
for f in filters:
    row = '**' + f["name"] + '**\n\n' + component(
        'aardvark', 'rangeslider',
        min=f["min"], max=f["max"], defaultValue=f["default"],
        color=f["color"], labelAlwaysOn=True,
    )
    controls += component('aardvark', 'card', variant='plain', children=row)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=controls))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description
`min`	number	Lower bound of the range (default `0`).
`max`	number	Upper bound of the range (default `100`).
`step`	number	Increment per move.
`defaultValue`	`'low, high'` string	Starting low and high pair (e.g. `'20, 80'`) — the reader can drag both.
`minRange`	number	Smallest allowed gap between the two thumbs.
`marks`	JSON array string	Tick marks as `[{"value": n, "label": "…"}, …]`.
`color`	theme color	Fill color of the span between the thumbs.
`size`	`xs–xl`	Track thickness.
`radius`	`xs–xl`	Corner rounding of the track.
`label`	string	Static text for the thumb tips.
`labelAlwaysOn`	boolean	Keep the value bubbles visible, not only while dragging.

`inverted`	boolean	Fill the outside instead of the inside.
`restrictToMarks`	boolean	Snap the thumbs to the `marks` positions only.
`showLabelOnHover`	boolean	Show the labels on hover (on by default; set `false` to suppress).
`disabled`	boolean	Make the range read-only.
`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

CSS Selectors

Target a `{% rangeslider %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every RangeSlider instance on the page */
[data-aardvark-island="RangeSlider"] { }

/* Mantine Styles API parts */
.mantine-RangeSlider-root { }
.mantine-RangeSlider-track { }
.mantine-RangeSlider-bar { }
.mantine-RangeSlider-thumb { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so dragging either thumb reads the range values, logs them to the console, and alerts them:

Source: Markdown

```
{% rangeslider min=0 max=100 defaultValue='20, 80' attr={'onchange': '''
const controls = Array.from(this.querySelectorAll('[role="slider"], input'));
const value = controls.map((control) => control.value ||
control.getAttribute('aria-valuenow')).filter(Boolean).join(' - ');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'rangeslider', min=0, max=100, defaultValue='20, 80',
attr={'onchange': '''
```

```
const controls = Array.from(this.querySelectorAll('[role="slider"], input'));
const value = controls.map((control) => control.value ||
control.getAttribute('aria-valuenow')).filter(Boolean).join(' - ');
console.log('attr demo value:', value);
alert(value);
'''})
```

Rating

A star rating for collecting feedback or showing a score. Set `defaultValue` (the stars shown on load — a float, so half-stars work with `fractions`), `count` (number of symbols), and `fractions` (sub-symbol granularity). Add `readOnly` to show a fixed score, or leave `defaultValue` off for an empty rating the reader fills in.

Use it as `{% rating %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'rating', …)`.

Default value

`defaultValue` is the number of stars shown on load. Leave it off for an empty rating.

Rating: ★★★☆☆ (3/5)

Source: Markdown

```
{% rating defaultValue=3 %}
```

Source: Python

```
component('aardvark', 'rating', defaultValue=3)
```

Fractions, count, color, and size

`fractions` adds sub-symbol steps (`2` gives half-stars), so a `defaultValue` of `2.5` lands on a half. `count` sets the number of symbols; `color` and `size` style them.

Rating: ★★☆☆☆ (2/5)

Source: Markdown

```
{% rating defaultValue=2.5 fractions=2 count=5 color="yellow" size="lg" %}
```

Source: Python

```
component('aardvark', 'rating', defaultValue=2.5, fractions=2, count=5,
          color='yellow', size='lg')
```

Read-only score

`readOnly` makes the rating display-only — the reader can't change it. Use it to show a score you've computed.

Rating: ★★★★☆ (4/5)

Source: Markdown

```
{% rating defaultValue=4 readOnly=true %}
```

Source: Python

```
component('aardvark', 'rating', defaultValue=4, readOnly=True)
```

With other components

Pair a read-only rating with text to show an average score inline, or an interactive one with a heading to prompt feedback.

Average customer rating:

Rating: ★★★★☆ (4/5)

How was this page?

Rating: ☆☆☆☆☆ (0/5)

Source: Markdown

```
Average customer rating: {% rating defaultValue=4.5 fractions=2 readOnly=true %}

How was this page?

{% rating count=5 name='page-helpfulness' %}
```

Source: Python

```
component('aardvark', 'rating', defaultValue=4.5, fractions=2, readOnly=True)
component('aardvark', 'rating', count=5, name='page-helpfulness')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`defaultValue`	number (float OK)	Stars selected on load. A float, so `2.5` works with `fractions`.
`count`	integer	Number of symbols (default `5`).

`fractions`	integer	Sub-symbol steps — `2` gives half-stars, `4` quarters (default `1`).
`color`	theme color name (`yellow`, `blue`, …)	Color of the filled symbols.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Overall size.
`readOnly`	bare flag (`true`)	Display-only; the reader can't change it.
`name`	string	Radio-group name — set a stable one if a page has several ratings.
`attr`	dict (`attr={…}`)	Raw HTML attributes (e.g. `onchange`) applied to the rendered root.

CSS Selectors

Target a `{% rating %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Rating instance on the page */
[data-aardvark-island="Rating"] { }

/* Mantine Styles API parts */
.mantine-Rating-root { }
.mantine-Rating-symbolGroup { }
.mantine-Rating-starSymbol { }
```

Injecting Attributes

`attr={…}` forwards a dict of raw HTML attributes — including inline event handlers — straight onto the widget's rendered root. It's the same mechanism every component supports: it can't ride the tag as a single scalar like `color="yellow"`, so it gets its own `attr={…}`. A rating is only useful if you can read the result, so here's the canonical use — hang an `onchange` on it and send each selection to Google Analytics:

Rating: ☆☆☆☆☆ (0/5)

Source: Markdown

```
{% rating count=5 attr={'onchange': '''
  (() => {
    if (this.dataset.rated) return;              // fire once, even if they re-pick
    this.dataset.rated = '1';
```

```
    const rating = Number(event.target.value);    // the star the reader picked
    window.gtag && gtag('event', 'rating_submitted', {
      event_label: 'docs-helpfulness',
      value: rating,
    });
  })()
'''} %}
```

Source: Python

```
component('aardvark', 'rating', count=5, attr={'onchange': '''
  (() => {
    if (this.dataset.rated) return;
    this.dataset.rated = '1';
    const rating = Number(event.target.value);
    window.gtag && gtag('event', 'rating_submitted', {
      event_label: 'docs-helpfulness',
      value: rating,
    });
  })()
'''})
```

Reading the value. A `change` from any star bubbles up to the root the handler sits on, so `event.target.value` is the chosen rating. (`this` inside the handler is that root element, not the value — if you'd rather go through it, `Number(this.querySelector('input:checked').value)` works too.) `Number(…)` also handles fractional values like `2.5`.

What makes this a good event:

- **`'rating_submitted'`** — a clear, present-tense GA4 event name, not a generic `click`, so it reads well in reports and the Realtime view.
- **`event_label: 'docs-helpfulness'`** — a stable, human-readable label naming which rating. Give each widget its own (`docs-helpfulness`, `pricing-clarity`, …) so several ratings stay distinct instead of blurring into one. (In GA4 `event_label` is a custom parameter — register it as a custom dimension in your property to see it in reports; `value` is built in. We skip Universal Analytics' `event_category` — GA4 doesn't use it.)
- **`value`** — the stars the reader chose, as a number GA4 can average.
- **`window.gtag &&`** — a guard so the handler is a no-op when GA isn't on the page (local preview, a consent gate) rather than throwing.
- **fires once** — `onchange` runs on every selection, so the `this.dataset.rated` flag sends the event on the first pick only; changing 3 → 5 stars won't send two events. (The default theme's page-feedback widget guards the same way.)

A site can lock handlers down with an `attrPolicy` in `aardvark.config.yaml` (e.g. `deny: ['on*']`).

RichTextEditor

A **live rich-text editor** you drop straight into a page: a formatting toolbar above an editable document area. The toolbar buttons reflect and toggle the current selection's formatting — **bold**, italic, underline — turn the current block into a bullet list, and attach or remove a link. Typing, selecting, and the usual keyboard shortcuts (`Cmd`/`Ctrl`+`B`, `+I`, `+U`) all drive the same document.

The editor is a runtime object — a Tiptap instance built from `useEditor()`, with its own document model and command chain — so it can't be produced from Markdown attributes the way a plain field is. It ships instead as a **project snippet** (`snippets/RichTextEditor.jsx`), which Aardvark exposes as the `{% RichTextEditor %}` tag automatically. It renders a placeholder during the build and hydrates into the interactive editor in the browser.

A live editor

`content` seeds the initial document. Pass an HTML string; Tiptap parses it into the document model, so inline formatting in the seed survives into the editable view.

Source: Markdown

```
{% RichTextEditor content='<p>Hello <strong>world</strong></p>' %}
```

Source: Python

```
component('RichTextEditor', content='<p>Hello <strong>world</strong></p>')
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`content`	HTML string (e.g. `<p>Hi <strong>there</strong></p>`)	Seeds the initial document; parsed into the editor's model client-side. Defaults to an empty document.
`attr`	`{ ... }` (a dict)	Raw HTML attributes injected onto the editor's root element — see Injecting Attributes.

CSS Selectors

The tag emits a hydration island wrapper, and the editor inside carries Mantine's `RichTextEditor` Styles API class names. Target the wrapper to position or frame the whole widget, and the inner classes to restyle the toolbar or the editable area:

```
/* The island wrapper the tag emits (the snippet's outermost node). */
[data-aardvark-island="RichTextEditor"] {
  max-width: 640px;
}

/* The editor root, toolbar, and editable content area. */
.mantine-RichTextEditor-root {
  border-radius: var(--mantine-radius-md);
}
.mantine-RichTextEditor-toolbar {
  /* the sticky bar of formatting controls */
}
.mantine-RichTextEditor-content {
  min-height: 200px; /* the editable document area */
}
```

Injecting Attributes

`attr` forwards a dict of raw HTML attributes onto the editor's **root** element. The snippet forwards its `ref` to that node, so anything in `attr` — `id`, `data-*`, `style`, an inline handler — lands directly on the `.mantine-RichTextEditor-root` element. Here it is wired to `onchange`, so when an edit is committed on blur, the handler reads the editor HTML, logs it to the console, and alerts it:

Source: Markdown

```
{% RichTextEditor content='<p>x</p>' attr={'onchange': '''
const editor = event.target.closest('[contenteditable="true"]') ||
this.querySelector('[contenteditable="true"]');
const value = this.value || this.dataset.aardvarkValue || (editor ? editor.innerHTML : '');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('RichTextEditor', content='<p>x</p>', attr={'onchange': '''
const editor = event.target.closest('[contenteditable="true"]') ||
this.querySelector('[contenteditable="true"]');
const value = this.value || this.dataset.aardvarkValue || (editor ? editor.innerHTML : '');
console.log('attr demo value:', value);
alert(value);
'''})
```

SegmentedControl

A segmented control — a row of mutually exclusive options, like a compact set of tabs. Give `data` a comma-separated list of labels (or a JSON array of `{label, value}` objects), and pick the starting option with `defaultValue`. The control is interactive, so readers can switch between segments.

Use it as `{% segmentedcontrol %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'segmentedcontrol', …)`.

The data list and default value

`data` is a comma-separated list of labels; `defaultValue` is the one selected on load.

Source: Markdown

```
{% segmentedcontrol data='Preview, Code, Export' defaultValue='Code' %}
```

Source: Python

```
component('aardvark', 'segmentedcontrol', data='Preview, Code, Export',
          defaultValue='Code')
```

From Python you can pass `data` as a real list of strings or `{label, value}` dicts — handy when building it in a loop:

Source: Python

```
component('aardvark', 'segmentedcontrol',
          data=['Preview', 'Code', 'Export'], defaultValue='Code')
```

Separate label and value

When the value you store differs from the label, pass a **JSON array** of `{label, value}` objects.

Source: Markdown

```
{% segmentedcontrol data='[{"label": "On", "value": "1"}, {"label": "Off", "value": "0"}]'
defaultValue='1' color='teal' %}
```

Source: Python

```
component('aardvark', 'segmentedcontrol',
          data=[{'label': 'On', 'value': '1'}, {'label': 'Off', 'value': '0'}],
          defaultValue='1', color='teal')
```

Colors, sizes, and radius

`color` tints the active indicator; `size` is `xs`–`xl`; `radius` rounds the track.

Source: Markdown

```
{% segmentedcontrol data='Light, Dark' color='grape' defaultValue='Dark' %}

{% segmentedcontrol data='S, M, L' size='lg' radius='xl' defaultValue='M' %}
```

Source: Python

```
component('aardvark', 'segmentedcontrol', data='Light, Dark', color='grape',
          defaultValue='Dark')
component('aardvark', 'segmentedcontrol', data='S, M, L', size='lg', radius='xl',
          defaultValue='M')
```

Full width, orientation, and states

`fullWidth` stretches to the container; `orientation='vertical'` stacks the segments; `disabled` greys it out; `readOnly` shows a selection the reader can't change; `withItemsBorders=false` drops the dividers between segments.

Source: Markdown

```
{% segmentedcontrol data='Day, Week, Month, Year' fullWidth=true defaultValue='Week' %}

{% segmentedcontrol data='Top, Middle, Bottom' orientation='vertical' defaultValue='Middle'
%}

{% segmentedcontrol data='Read, Write' defaultValue='Read' disabled=true %}

{% segmentedcontrol data='One, Two, Three' defaultValue='Two' withItemsBorders=false %}
```

Source: Python

```
component('aardvark', 'segmentedcontrol', data='Day, Week, Month, Year',
          fullWidth=True, defaultValue='Week')
component('aardvark', 'segmentedcontrol', data='Top, Middle, Bottom',
          orientation='vertical', defaultValue='Middle')
component('aardvark', 'segmentedcontrol', data='Read, Write', defaultValue='Read',
          disabled=True)
component('aardvark', 'segmentedcontrol', data='One, Two, Three', defaultValue='Two',
          withItemsBorders=False)
```

With other components

Pair a segmented control with a heading and a button to build a small view-switcher row.

Choose a view

Apply

Source: Markdown

```
Choose a view

{% segmentedcontrol data='List, Grid, Calendar' defaultValue='Grid' fullWidth=true %}

{% button %}Apply{% endButton %}
```

Source: Python

```
component('aardvark', 'segmentedcontrol', data='List, Grid, Calendar',
          defaultValue='Grid', fullWidth=True)
component('aardvark', 'button', children='Apply')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`data`	comma-separated labels (`One, Two, Three`) or a JSON array of `{label, value}`	The options. A bare label list uses each label as its own value; from Python, also a real list of strings or dicts.
`defaultValue`	one of the option values	Which option starts selected.
`color`	theme color name (`blue`, `teal`, …)	Color of the active indicator.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Overall size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner rounding of the track.
`orientation`	`horizontal` (default), `vertical`	Lay the segments in a row or a column.
`fullWidth`	bare flag (`true`)	Stretch to the container’s width.

`disabled`	bare flag (`true`)	Render non-interactive.
`readOnly`	bare flag (`true`)	Display-only — show a selection the reader can't change.
`withItemsBorders`	bare flag — `false` to turn off (on by default)	Dividers between items.
`attr`	dict (`attr={…}`)	Raw HTML attributes (e.g. `onchange`) applied to the rendered root.

CSS Selectors

Target a `{% segmentedcontrol %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every SegmentedControl instance on the page */
[data-aardvark-island="SegmentedControl"] { }

/* Mantine Styles API parts */
.mantine-SegmentedControl-root { }
.mantine-SegmentedControl-control { }
.mantine-SegmentedControl-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so selecting a segment logs the active value to the console and alerts it:

Source: Markdown

```
{% segmentedcontrol data='Preview, Code, Export' defaultValue='Code' attr={'onchange': '''
const value = this.querySelector('input:checked')?.value ?? '(none)';
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'segmentedcontrol', data='Preview, Code, Export', defaultValue='Code',
attr={'onchange': '''
const value = this.querySelector('input:checked')?.value ?? '(none)';
console.log('attr demo value:', value);
alert(value);
'''})
```

Slider

A **built-in** tag for a slider — let the reader pick a number within a range by dragging a thumb. It ships with Aardvark, so it's a single tag with no setup. Tune the bounds with `min` and `max`, the increment with `step`, and the starting position with `defaultValue`.

Use it as `{% slider %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'slider', …)`.

Demonstrations

Basic range

Set `min`, `max`, `step`, and a `defaultValue`. Drag the thumb to change it.

Source: Markdown

```
{% slider min=0 max=100 step=5 defaultValue=40 %}
```

Source: Python

```
component('aardvark', 'slider', min=0, max=100, step=5, defaultValue=40)
```

Marks

Pass `marks` as a JSON array of `{value, label}` objects. Add `restrictToMarks` to snap the thumb to those positions only.

Source: Markdown

```
{% slider min=0 max=100 defaultValue=50 color='grape' marks='[{"value": 0, "label": "0%"},
{"value": 50, "label": "50%"}, {"value": 100, "label": "100%"}]' %}
```

Source: Python

The `marks` array rides through as a native Python list of dicts:

```
component('aardvark', 'slider',
          min=0, max=100, defaultValue=50, color='grape',
          marks='[{"value": 0, "label": "0%"}, {"value": 50, "label": "50%"}, {"value": 100,
"label": "100%"}]')
```

Colors, sizes, and inverted fill

Any theme `color` paints the filled track; `size` and `radius` take `xs`–`xl`. Add `inverted` to fill from the max side instead of the min, and `labelAlwaysOn` to keep the value bubble visible.

Source: Markdown

```
{% slider defaultValue=30 color='teal' size='lg' labelAlwaysOn=true %}
{% slider defaultValue=70 color='orange' radius='xl' inverted=true %}
```

Source: Python

```
component('aardvark', 'slider', defaultValue=30, color='teal', size='lg',
labelAlwaysOn=True)
component('aardvark', 'slider', defaultValue=70, color='orange', radius='xl', inverted=True)
```

Disabled

Add the bare `disabled` flag to make the slider read-only.

Source: Markdown

```
{% slider defaultValue=60 disabled=true %}
```

Source: Python

```
component('aardvark', 'slider', defaultValue=60, disabled=True)
```

With other components

Build a labelled control inside a `{% card %}`, and generate a whole row of sliders from data with a Python loop — the same `component('aardvark', 'slider', …)` call, once per item.

Bass

Mid

Treble

Source: Python

```
levels = [
    {"name": "Bass", "value": 70, "color": "blue"},
    {"name": "Mid", "value": 45, "color": "grape"},
    {"name": "Treble", "value": 60, "color": "teal"},
]
mixer = ''
```

```
for level in levels:
    row = '**' + level["name"] + '**\n\n' + component(
        'aardvark', 'slider',
        min=0, max=100, defaultValue=level["value"],
        color=level["color"], labelAlwaysOn=True,
    )
    mixer += component('aardvark', 'card', variant='plain', children=row)
page.print(component('aardvark', 'cardGrid', cols={'base': 1, 'sm': 3}, children=mixer))
```

Attributes

Omit any attribute to take its default.

Attribute	Values	Description
`min`	number	Lower bound of the range (default `0`).
`max`	number	Upper bound of the range (default `100`).
`step`	number	Increment per move.
`defaultValue`	number	Starting value — the reader can drag it.
`marks`	JSON array string	Tick marks as `[{"value": n, "label": "…"}, …]`.
`color`	theme color	Fill color of the track (e.g. `blue`, `grape`, `teal`).
`size`	`xs`–`xl`	Track thickness.
`radius`	`xs`–`xl`	Corner rounding of the track.
`label`	string	Static text for the thumb tip (a plain string sets a fixed label).
`labelAlwaysOn`	boolean	Keep the value bubble visible, not only while dragging.
`inverted`	boolean	Fill from the max side instead of the min.
`restrictToMarks`	boolean	Snap the thumb to the `marks` positions only.
`showLabelOnHover`	boolean	Show the label on hover (on by default; set `false` to suppress).
`disabled`	boolean	Make the slider read-only.

`attr`	raw-HTML map	Extra HTML attributes, passed as `attr={…}`.

CSS Selectors

Target a `{% slider %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Slider instance on the page */
[data-aardvark-island="Slider"] { }

/* Mantine Styles API parts */
.mantine-Slider-root { }
.mantine-Slider-track { }
.mantine-Slider-bar { }
.mantine-Slider-thumb { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so dragging the slider reads the changed control value, logs it to the console, and alerts it:

Source: Markdown

```
{% slider min=0 max=100 step=5 defaultValue=40 attr={'onchange': '''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'slider', min=0, max=100, step=5, defaultValue=40, attr={'onchange':
'''
const control = event.target.closest('[role="slider"], input') ||
this.querySelector('[role="slider"], input');
const value = this.value || this.dataset.aardvarkValue || (control ? (control.value ||
control.getAttribute('aria-valuenow')) : '');
console.log('attr demo value:', value);
alert(value);
'''})
```

Switch

A toggle switch with a label. The label is the `label` attribute or the block body, and the switch is interactive — readers can flip it. Add `defaultChecked` to start it on, and `onLabel` / `offLabel` for short text shown inside the track.

Use it as `{% switch %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'switch', …)`.

Label, default state, and track text

The label comes from the `label` attribute or the block body. `defaultChecked` starts it on; `onLabel` / `offLabel` print short text inside the track.

Dark mode

Notifications

Source: Markdown

```
{% switch label='Dark mode' defaultChecked=true %}

{% switch label='Notifications' onLabel='ON' offLabel='OFF' %}
```

Source: Python

```
component('aardvark', 'switch', label='Dark mode', defaultChecked=True)

component('aardvark', 'switch', label='Notifications', onLabel='ON', offLabel='OFF')
```

Colors and sizes

`color` is any theme color for the on-state track; `size` is `xs`–`xl`.

blue

teal

large

Source: Markdown

```
{% switch label='blue' defaultChecked=true %}
{% switch label='teal' color='teal' defaultChecked=true %}
{% switch label='large' size='lg' defaultChecked=true %}
```

Source: Python

```
component('aardvark', 'switch', label='blue', defaultChecked=True)
```

```
component('aardvark', 'switch', label='teal', color='teal', defaultChecked=True)
component('aardvark', 'switch', label='large', size='lg', defaultChecked=True)
```

Radius, thumb indicator, label position, and disabled

`radius` rounds the track; `withThumbIndicator` shows the inner thumb dot; `labelPosition='left'` puts the label first; `disabled` makes it non-interactive.

Square track

Thumb indicator

Label on the left

Disabled

Source: Markdown

```
{% switch label='Square track' radius='xs' defaultChecked=true %}
{% switch label='Thumb indicator' withThumbIndicator=true defaultChecked=true %}
{% switch label='Label on the left' labelPosition='left' defaultChecked=true %}
{% switch label='Disabled' disabled=true defaultChecked=true %}
```

Source: Python

```
component('aardvark', 'switch', label='Square track', radius='xs', defaultChecked=True)
component('aardvark', 'switch', label='Thumb indicator', withThumbIndicator=True,
          defaultChecked=True)
component('aardvark', 'switch', label='Label on the left', labelPosition='left',
          defaultChecked=True)
component('aardvark', 'switch', label='Disabled', disabled=True, defaultChecked=True)
```

Helper text

`description` and `error` add helper text below the label.

Beta features

Required setting

Source: Markdown

```
{% switch label='Beta features' description='Experimental — may change without notice.' %}
{% switch label='Required setting' error='Turn this on to save.' %}
```

Source: Python

```
component('aardvark', 'switch', label='Beta features',
          description='Experimental — may change without notice.')
component('aardvark', 'switch', label='Required setting', error='Turn this on to save.')
```

With other components

Combine a switch with a heading and a button to build a small settings row.

Notification settings

Email me about replies

Save

Source: Markdown

```
Notification settings

{% switch label='Email me about replies' defaultChecked=true %}

{% button %}Save{% endButton %}
```

Source: Python

```
component('aardvark', 'switch', label='Email me about replies', defaultChecked=True)
component('aardvark', 'button', children='Save')
```

Attributes

Omit any attribute to take its Mantine default. For a React icon on the thumb (`thumbIcon`), call `{% component('Switch', …) %}` directly — a build-time tag can only pass the on/off labels as text.

Attribute	Valid values	Description
`label`	string	The label, when not using the block body. Plain text — Markdown (e.g. `**bold**`) is not rendered.
`defaultChecked`	bare flag (`true`)	Start the switch on. It stays interactive — the reader can toggle it.
`onLabel`	string	Short text shown inside the track in the on position.
`offLabel`	string	Short text shown inside the track in the off position.
`disabled`	bare flag (`true`)	Render non-interactive.

`color`	theme color name (`blue`, `teal`, ...)	Track color in the on state.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Overall size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner rounding of the track.
`labelPosition`	`right` (default), `left`	Which side of the switch the label sits on.
`withThumbIndicator`	bare flag (`true`)	Show the inner thumb indicator dot.
`description`	string	Helper text shown below the label.
`error`	string	Error text shown below the label (also marks the switch invalid).
`attr`	dict (`attr={…}`)	Raw HTML attributes (e.g. `onchange`) applied to the rendered root.

CSS Selectors

Target a `{% switch %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Switch instance on the page */
[data-aardvark-island="Switch"] { }

/* Mantine Styles API parts */
.mantine-Switch-root { }
.mantine-Switch-track { }
.mantine-Switch-thumb { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so toggling it logs its checked state to the console and alerts it:

Dark mode

Source: Markdown

```
{% switch label='Dark mode' defaultChecked=true attr={'onchange': '''
```

```
const value = this.checked;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'switch', label='Dark mode', defaultChecked=True, attr={'onchange':
'''
const value = this.checked;
console.log('attr demo value:', value);
alert(value);
'''})
```

Textarea

A multi-line text field. It carries the same Input wrapper as TextInput — label, description, error, sections — and adds the textarea specifics: `autosize`, `minRows`/`maxRows`, and `resize`. Reach for it for bios, notes, messages, and any longer free-form value.

Use it as `{% textarea %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'textarea', …)`.

Basic field

A label, a placeholder, and a fixed starting height with `minRows`.

Bio

Source: Markdown

```
{% textarea label='Bio' placeholder='Tell us about yourself' minRows=3 %}
```

Source: Python

```
component('aardvark', 'textarea', label='Bio',
          placeholder='Tell us about yourself', minRows=3)
```

Autosize

`autosize` grows the field with its content. Set a floor with `minRows` and a cap with `maxRows`.

Notes

Source: Markdown

```
{% textarea label='Notes' autosize=true minRows=2 maxRows=6 placeholder='Grows as you type,
up to 6 rows' %}
```

Source: Python

```
component('aardvark', 'textarea', label='Notes', autosize=True,
          minRows=2, maxRows=6,
          placeholder='Grows as you type, up to 6 rows')
```

Resize

`resize` is `none`, `both`, `horizontal`, or `vertical` — it controls the drag handle in the field's corner.

Free resize

Vertical only

Source: Markdown

```
{% textarea label='Free resize' resize='both' placeholder='Drag the corner in any direction'
%}

{% textarea label='Vertical only' resize='vertical' placeholder='Drag the corner up and
down' %}
```

Source: Python

```
component('aardvark', 'textarea', label='Free resize', resize='both',
          placeholder='Drag the corner in any direction')

component('aardvark', 'textarea', label='Vertical only', resize='vertical',
          placeholder='Drag the corner up and down')
```

Required, asterisk, and disabled

`required` and `withAsterisk` add the asterisk; `disabled` greys the control out.

Message

Imported note

Locked note

Source: Markdown

```
{% textarea label='Message' required=true placeholder='Required field' minRows=2 %}

{% textarea label='Imported note' withAsterisk=true defaultValue='From the legacy system'
minRows=2 %}

{% textarea label='Locked note' disabled=true defaultValue='Read-only' minRows=2 %}
```

Source: Python

```
component('aardvark', 'textarea', label='Message', required=True,
          placeholder='Required field', minRows=2)

component('aardvark', 'textarea', label='Imported note', withAsterisk=True,
          defaultValue='From the legacy system', minRows=2)

component('aardvark', 'textarea', label='Locked note', disabled=True,
          defaultValue='Read-only', minRows=2)
```

Error

`error` shows a validation message below the field and switches it to the error color.

Bio

Source: Markdown

```
{% textarea label='Bio' error='Keep it under 280 characters' minRows=2 defaultValue='A very long bio that has gone over the limit…' %}
```

Source: Python

```
component('aardvark', 'textarea', label='Bio',
          error='Keep it under 280 characters', minRows=2,
          defaultValue='A very long bio that has gone over the limit…')
```

Variants, size, and radius

`variant` is `default`, `filled`, or `unstyled`; `size` and `radius` take `xs`–`xl`.

Filled

Large, round

Source: Markdown

```
{% textarea label='Filled' variant='filled' placeholder='filled variant' minRows=2 %}

{% textarea label='Large, round' size='lg' radius='lg' placeholder='lg size, lg radius' minRows=2 %}
```

Source: Python

```
component('aardvark', 'textarea', label='Filled', variant='filled',
          placeholder='filled variant', minRows=2)

component('aardvark', 'textarea', label='Large, round', size='lg', radius='lg',
          placeholder='lg size, lg radius', minRows=2)
```

Sections

`leftSection` and `rightSection` take text shown inside the field.

Comment

Source: Markdown

```
{% textarea label='Comment' leftSection='⎚' placeholder='Leave a comment' minRows=2 %}
```

Source: Python

```
component('aardvark', 'textarea', label='Comment', leftSection='⎚',
          placeholder='Leave a comment', minRows=2)
```

With other components

Pair a textarea with a TextInput inside a {% card %} to build a feedback form.

Send feedback

Subject

Details

Source: Markdown

```
{% card title='Send feedback' %}
{% textinput label='Subject' placeholder='What is this about?' required=true %}

{% textarea label='Details' autosize=true minRows=3 maxRows=8 placeholder='Tell us more…' %}
{% endCard %}
```

Source: Python

```
subject = component('aardvark', 'textinput', label='Subject',
                    placeholder='What is this about?', required=True)
details = component('aardvark', 'textarea', label='Details', autosize=True,
                    minRows=3, maxRows=8, placeholder='Tell us more…')
component('aardvark', 'card', title='Send feedback', children=subject + details)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when the field is empty.

`error`	string	Validation message below the field; switches it to the error color.
`defaultValue`	string	Initial value of the field.
`variant`	`default`, `filled`, `unstyled`	Visual style of the input.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS length	Corner radius.
`required`	bool (`true` / `false`)	Mark the field required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required` attribute.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`autosize`	bool (`true` / `false`)	Grow the field with its content.
`minRows`	integer	Minimum number of visible rows (a floor when `autosize` is on).
`maxRows`	integer	Maximum number of rows the field grows to (with `autosize`).
`resize`	`none`, `both`, `horizontal`, `vertical`	Direction the user can resize the field.
`leftSection`	string	Text shown inside the field, before the value.
`rightSection`	string	Text shown inside the field, after the value.

CSS Selectors

Target a `{% textarea %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every Textarea instance on the page */
[data-aardvark-island="Textarea"] { }

/* Mantine Styles API parts */
.mantine-Textarea-root { }
.mantine-Textarea-input { }
```

```
.mantine-Textarea-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Bio

Source: Markdown

```
{% textarea label='Bio' placeholder='Tell us about yourself' minRows=3 attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'textarea', label='Bio', placeholder='Tell us about yourself',
minRows=3, attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

TextInput

A single-line text field with the full Input wrapper: a label, helper description, error message, and left/right sections. Reach for it for emails, names, URLs, search boxes — any short, single-line value. Every field below takes a `label` so the control stays accessible.

Use it as `{% textinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'textinput', …)`.

Basic field

A label, a placeholder, and a description below the label.

Email

Source: Markdown

```
{% textinput label='Email' placeholder='you@example.com' description='Work address' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Email',
          placeholder='you@example.com', description='Work address')
```

Required and asterisk

`required` marks the field required and adds the asterisk; `withAsterisk` adds the asterisk without the HTML `required` attribute.

Username

Display name

Source: Markdown

```
{% textinput label='Username' required=true placeholder='Pick a handle' %}

{% textinput label='Display name' withAsterisk=true placeholder='Shown publicly' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Username', required=True,
          placeholder='Pick a handle')

component('aardvark', 'textinput', label='Display name', withAsterisk=True,
          placeholder='Shown publicly')
```

Error and disabled

`error` shows a validation message below the field and switches it to the error color; `disabled` greys the control out.

Email

Account ID

Source: Markdown

```
{% textinput label='Email' error='Enter a valid email address' placeholder='you@example.com'
%}

{% textinput label='Account ID' disabled=true defaultValue='acct_8842' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Email',
          error='Enter a valid email address', placeholder='you@example.com')

component('aardvark', 'textinput', label='Account ID', disabled=True,
          defaultValue='acct_8842')
```

Input type

`type` sets the HTML input type — `email`, `url`, `tel`, `search`, `password`, `number`, and so on — which drives browser validation and the mobile keyboard.

Website

Phone

Source: Markdown

```
{% textinput label='Website' type='url' placeholder='https://example.com' %}

{% textinput label='Phone' type='tel' placeholder='+1 555 0100' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Website', type='url',
          placeholder='https://example.com')

component('aardvark', 'textinput', label='Phone', type='tel',
          placeholder='+1 555 0100')
```

Variants

`variant` is `default`, `filled`, or `unstyled`.

Default

Filled

Unstyled

Source: Markdown

```
{% textinput label='Default' variant='default' placeholder='default variant' %}

{% textinput label='Filled' variant='filled' placeholder='filled variant' %}

{% textinput label='Unstyled' variant='unstyled' placeholder='unstyled variant' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Default', variant='default',
          placeholder='default variant')

component('aardvark', 'textinput', label='Filled', variant='filled',
          placeholder='filled variant')

component('aardvark', 'textinput', label='Unstyled', variant='unstyled',
          placeholder='unstyled variant')
```

Size and radius

`size` takes `xs`–`xl`; `radius` takes `xs`–`xl` (or any CSS length).

Extra small

Large, round

Source: Markdown

```
{% textinput label='Extra small' size='xs' placeholder='size xs' %}

{% textinput label='Large, round' size='lg' radius='xl' placeholder='lg / xl radius' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Extra small', size='xs',
          placeholder='size xs')

component('aardvark', 'textinput', label='Large, round', size='lg', radius='xl',
          placeholder='lg / xl radius')
```

Sections

`leftSection` and `rightSection` take text (an emoji, symbol, or short string) shown inside the field, before and after the value.

Website

Source: Markdown

```
{% textinput label='Website' leftSection='@' rightSection='.com' placeholder='your-site' %}
```

Source: Python

```
component('aardvark', 'textinput', label='Website', leftSection='@',
          rightSection='.com', placeholder='your-site')
```

With other components

Drop fields into a {% card %} to build a compact sign-in form.

Sign in

Email

Workspace

Source: Markdown

```
{% card title='Sign in' %}
{% textinput label='Email' type='email' placeholder='you@example.com' required=true %}

{% textinput label='Workspace' leftSection='@' placeholder='acme' %}
{% endCard %}
```

Source: Python

```
email = component('aardvark', 'textinput', label='Email', type='email',
                  placeholder='you@example.com', required=True)
workspace = component('aardvark', 'textinput', label='Workspace',
                      leftSection='@', placeholder='acme')
component('aardvark', 'card', title='Sign in', children=email + workspace)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	Field label above the input.
`description`	string	Helper text below the label.
`placeholder`	string	Placeholder shown when the field is empty.
`error`	string	Validation message below the field; switches it to the error color.
`type`	`text`, `email`, `url`, `tel`, `search`, `password`, `number`, …	HTML input type — drives browser validation and the mobile keyboard.
`defaultValue`	string	Initial value of the field.
`variant`	`default`, `filled`, `unstyled`	Visual style of the input.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS length	Corner radius.
`required`	bool (`true` / `false`)	Mark the field required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required` attribute.
`disabled`	bool (`true` / `false`)	Render the field disabled.
`leftSection`	string	Text shown inside the field, before the value.
`rightSection`	string	Text shown inside the field, after the value.

CSS Selectors

Target a `{% textinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every TextInput instance on the page */
[data-aardvark-island="TextInput"] { }

/* Mantine Styles API parts */
.mantine-TextInput-root { }
```

```
.mantine-TextInput-input { }
.mantine-TextInput-label { }
.mantine-TextInput-section { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Email

Source: Markdown

```
{% textinput label='Email' placeholder='you@example.com' description='Work address'
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'textinput', label='Email', placeholder='you@example.com',
description='Work address', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

TimeInput

A **time field** built on the browser's native `<input type="time">`, dressed in the Mantine Input wrapper. Reach for it for a plain time entry (a meeting start, an alarm). It hydrates into an interactive island.

Use it as `{% timeinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'timeinput', …)`.

Times are strings

The value is a **time string** — `HH:mm` (or `HH:mm:ss` with `withSeconds`).

A time field

`withSeconds` adds the seconds field; the usual Input wrapper props apply.

Stand-up

Precise start

Source: Markdown

```
{% timeinput label='Stand-up' defaultValue='09:30' description='24-hour clock' %}

{% timeinput label='Precise start' defaultValue='09:30:15' withSeconds=true %}
```

Source: Python

```
component('aardvark', 'timeinput', label='Stand-up', defaultValue='09:30',
          description='24-hour clock')

component('aardvark', 'timeinput', label='Precise start',
          defaultValue='09:30:15', withSeconds=True)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`defaultValue`	`HH:mm` string (or `HH:mm:ss` with `withSeconds`)	Initial value.
`withSeconds`	bool (`true` / `false`)	Show and capture seconds.
`label`	string	Field label above the input.

`description`	string	Helper text below the label.
`error`	string	Validation message; switches the field to the error color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Control size.
`radius`	`xs`–`xl` or a CSS length	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`required`	bool (`true` / `false`)	Mark required and add the asterisk.
`withAsterisk`	bool (`true` / `false`)	Add the asterisk without the HTML `required`.
`disabled`	bool (`true` / `false`)	Render the field disabled.

CSS Selectors

Target a `{% timeinput %}` from your own CSS with the island data attribute or the Mantine Styles API part classes:

```
/* Every TimeInput instance on the page */
[data-aardvark-island="TimeInput"] { }

/* Mantine Styles API parts */
.mantine-TimeInput-root { }
.mantine-TimeInput-input { }
.mantine-TimeInput-label { }
```

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes — including inline event handlers — straight onto the rendered element. Here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Stand-up

Source: Markdown

```
{% timeinput label='Stand-up' defaultValue='09:30' description='24-hour clock'
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
```

```
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'timeinput', label='Stand-up', defaultValue='09:30',
description='24-hour clock', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

Combobox

The **Combobox** family is Mantine's set of dropdown / select / autocomplete inputs. Aardvark wraps each one as a first-class `{% tag %}` so you can drop a searchable select or a tags field into a page with no JavaScript. They hydrate into fully interactive islands in the browser.

- Select — a searchable single-select dropdown.
- Autocomplete — a free-text input with a suggestion list.
- MultiSelect — pick several options, shown as removable pills.
- TagsInput — type free-text tags, with optional suggestions.
- TreeSelect — pick a leaf from a collapsible hierarchy.
- Pill — the small rounded chip the multi-value inputs render.
- PillsInput — the input shell that holds pills plus a field.
- Combobox — the low-level primitive everything else is built on.

Options data

Every select-style tag takes its options through `data`. The simplest form is a comma-separated list of strings, and `value::label` pairs let a short stored value carry a friendly label:

```
{% select data='React, Vue, Svelte' %}
{% select data='us::United States, ca::Canada, mx::Mexico' %}
```

For grouped, disabled, or otherwise richer options, pass a full JSON array through `dataJson` instead (it wins over `data`):

```
{% select
dataJson='[{"group":"Frontend","items":["React","Vue"]},{"group":"Backend","items":["Django"
,"Rails"]}]' %}
```

`Select`, `Autocomplete`, `MultiSelect`, and `TagsInput` all share this convention; `TreeSelect` takes a nested JSON tree instead (see its page).

High-level vs. primitives

Most of the time you want the ready-made inputs — `Select`, `MultiSelect`, `Autocomplete`, `TagsInput`, `TreeSelect`. The remaining two, Combobox and PillsInput, are the **composition primitives** the others are built from; their pages explain when (rarely) you'd reach for them directly.

Autocomplete

A built-in tag for a free-text input with a suggestion dropdown. Unlike Select, the typed value need not match an option — the list is only a set of suggestions, and the reader can submit anything. It hydrates into a fully interactive Mantine input in the browser, with no JavaScript to write.

Use it as `{% autocomplete %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'autocomplete', …)`.

Basic autocomplete

A `label`, a `placeholder`, and a comma-separated `data` list of suggestions. Type freely — the suggestions filter as you go, but any value submits:

Email

Source: Markdown

```
{% autocomplete label='Email' placeholder='you@…' data='gmail.com, outlook.com, proton.me'
clearable=true %}
```

Source: Python

```
component('aardvark', 'autocomplete',
          label='Email', placeholder='you@…',
          data='gmail.com, outlook.com, proton.me',
          clearable=True)
```

Grouped suggestions with dataJson

For grouped suggestions pass a full JSON array through `dataJson` — a list of `{group, items}` objects. `dataJson` wins over `data` when both are set:

Framework

Source: Markdown

```
{% autocomplete label='Framework'
dataJson='[{"group":"React","items":["Next.js","Remix"]},{"group":"Vue","items":["Nuxt"]}]'
%}
```

Source: Python

```
component('aardvark', 'autocomplete',
          label='Framework',
          dataJson='[{"group":"React","items":["Next.js","Remix"]},'
                   '{"group":"Vue","items":["Nuxt"]}]')
```

Sizes, variants, and a default value

`size` runs `xs`–`xl`; `variant` is `default`, `filled`, or `unstyled`. `defaultValue` seeds the field with text on load:

Host

Source: Markdown

```
{% autocomplete label='Host' data='localhost, 127.0.0.1, example.com'
defaultValue='localhost' size='md' variant='filled' radius='md' %}
```

Source: Python

```
component('aardvark', 'autocomplete',
          label='Host', data='localhost, 127.0.0.1, example.com',
          defaultValue='localhost', size='md', variant='filled', radius='md')
```

Required, with description and error

`description` adds helper text; `error` shows a validation message; `required` with `withAsterisk` marks the field and shows the asterisk:

Domain

Source: Markdown

```
{% autocomplete label='Domain' data='example.com, example.org' description='Where to send
mail' required=true withAsterisk=true %}
```

Source: Python

```
component('aardvark', 'autocomplete',
          label='Domain', data='example.com, example.org',
          description='Where to send mail',
          required=True, withAsterisk=True)
```

With other components

An autocomplete pairs well with other content inside a card — here a text lead-in above the field:

Pick or type an email provider.

Provider

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}Pick or type an email provider.{% endText %}

{% autocomplete label='Provider' placeholder='you@…' data='gmail.com, outlook.com,
proton.me, fastmail.com' clearable=true %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='Pick or type an email provider.', size='sm',
c='dimmed') +
    component('aardvark', 'autocomplete',
              label='Provider', placeholder='you@…',
              data='gmail.com, outlook.com, proton.me, fastmail.com',
              clearable=True)
))
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The body is ignored — `autocomplete` is a form control, not a container.

Attribute	Valid values	Description
`data`	Comma-separated list	The suggestions. These are plain strings — the typed value is free text, so there is no separate value/label.
`dataJson`	JSON array string	A full suggestions array, including `{group, items}` groups. Wins over `data` when both are set.
`label`	String	The field label.
`placeholder`	String	Empty-state text inside the input.
`description`	String	Helper text below the label.

`error`	String	Validation message; also styles the field red.
`defaultValue`	String	The text shown on load.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Input size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`clearable`	`true`, `false` (default `false`)	Show an × to clear the value.
`disabled`	`true`, `false` (default `false`)	Disable the input.
`required`	`true`, `false` (default `false`)	Mark the field required.
`withAsterisk`	`true`, `false` (default `false`)	Show the required asterisk on the label.
`withScrollArea`	`true`, `false` (default `true`)	Wrap a long suggestion list in a scroll area.
`maxDropdownHeight`	Integer (px)	Cap the dropdown height.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Autocomplete"]` — or through the Mantine Styles API classes (`.mantine-Autocomplete-root` and its inner parts):

```
/* Every rendered Autocomplete carries this island marker */
[data-aardvark-island="Autocomplete"] { }

/* Mantine Styles API class on the root element */
.mantine-Autocomplete-root { }
.mantine-Autocomplete-input { }
.mantine-Autocomplete-label { }
.mantine-Autocomplete-wrapper { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — here it is wired to `onchange`, so changing the field logs its new value to the console and alerts it:

Email

Source: Markdown

```
{% autocomplete label='Email' placeholder='you@…' data='gmail.com, outlook.com, proton.me'
clearable=true attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'autocomplete',
          label='Email', placeholder='you@…',
          data='gmail.com, outlook.com, proton.me',
          clearable=True, attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

Combobox

`Combobox` is the low-level composition primitive the entire select family is built on — Select, Autocomplete, MultiSelect, and TagsInput are all assembled from it. A real `Combobox` is wired up with the `useCombobox()` React hook — state a build-time Markdown tag can't supply — so this tag renders a small, self-contained working example of the primitive (a button-trigger single-select) so you can see it in action. For a configurable field reach for Select or one of the others; to build a fully custom dropdown, compose the primitive in a snippet (see Building your own below).

Use it as `{% combobox %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'combobox', …)`.

Working example

Give the example its options through `data` — a comma-separated list, or `value::label` pairs (a double colon, so a stored value carries a display label). `placeholder` is the empty-state label on the trigger:

Source: Markdown

```
{% combobox data='React, Vue, Svelte, Angular' placeholder='Pick a framework' %}
```

Source: Python

```
component('aardvark', 'combobox',
          data='React, Vue, Svelte, Angular',
          placeholder='Pick a framework')
```

value::label pairs and dataJson

`value::label` pairs give each option a stored value and a display label. For richer options pass a full JSON array through `dataJson`, which wins over `data`:

Source: Markdown

```
{% combobox data='us::United States, ca::Canada, mx::Mexico' placeholder='Pick a country' %}
{% combobox dataJson='["Small","Medium","Large"]' placeholder='Pick a size' %}
```

Source: Python

```
component('aardvark', 'combobox', data='us::United States, ca::Canada, mx::Mexico',
placeholder='Pick a country')
component('aardvark', 'combobox', dataJson='["Small","Medium","Large"]', placeholder='Pick a
size')
```

With other components

The example trigger sits inside a card alongside other content — here a text lead-in above it:

> A single-select built from the raw Combobox primitives.

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}A single-select built from the raw Combobox primitives.{%
endText %}

{% combobox data='React, Vue, Svelte, Angular' placeholder='Pick a framework' %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='A single-select built from the raw Combobox
primitives.', size='sm', c='dimmed') +
    component('aardvark', 'combobox', data='React, Vue, Svelte, Angular', placeholder='Pick
a framework')
))
```

Building your own

The `combobox` tag is a demonstration of the primitive, not a configurable field. When you need a genuinely custom dropdown — custom option rendering, custom filtering, a multi-value layout — drop a React component in `snippets/` and compose Mantine's `Combobox`, `Combobox.Target`, `Combobox.Options`, and `Combobox.Option` there, then call it from Markdown like any custom component. For anything short of that, the four ready-made inputs — Select, Autocomplete, MultiSelect, and TagsInput — already cover it.

Attributes

The tag renders a fixed demonstration trigger, so it takes only the option list and a placeholder. The body is ignored.

Attribute	Valid values	Description
`data`	Comma-separated list, or `value::label` pairs (double colon)	Options for the demo. A bare item is both value and label; `value::label` carries a separate stored value. A single `:` is left intact.

`dataJson`	JSON array string	A full options array — plain strings or `{value, label}` objects. Wins over `data` when both are set.
`placeholder`	String	The empty-state label on the trigger.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Combobox"]`. The `Combobox` primitive is a context provider with no DOM root of its own, so style its trigger through the `Input` parts and its popup through the Combobox Styles API parts:

```
/* Every rendered Combobox carries this island marker */
[data-aardvark-island="Combobox"] { }

/* The trigger is rendered through Mantine's Input */
.mantine-InputBase-input { }

/* Mantine Styles API parts on the dropdown popup */
.mantine-Combobox-dropdown { }
.mantine-Combobox-option { }
.mantine-Combobox-options { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% combobox data='React, Vue, Svelte, Angular' placeholder='Pick a framework'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'combobox',
          data='React, Vue, Svelte, Angular',
          placeholder='Pick a framework', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
```

```
'''})
```

MultiSelect

A built-in tag for picking several options at once. Each chosen value shows as a removable Pill inside the input, and the value is a list. Options come through `data` (the same convention as Select — a comma-separated list or `value::label` pairs). It hydrates into a fully interactive Mantine input in the browser.

Use it as `{% multiselect %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'multiselect', …)`.

Basic multiselect

A `label`, a `placeholder`, and a comma-separated `data` list. `searchable` lets the reader filter; `clearable` shows an × to drop all selections:

Stack

Source: Markdown

```
{% multiselect label='Stack' placeholder='Pick a few' data='React, Vue, Svelte, Angular,
Solid' searchable=true clearable=true %}
```

Source: Python

```
component('aardvark', 'multiselect',
          label='Stack', placeholder='Pick a few',
          data='React, Vue, Svelte, Angular, Solid',
          searchable=True, clearable=True)
```

value::label pairs with a default value

`value::label` pairs (a double colon) give each option a stored value and a display label. `defaultValue` is a comma-separated list of values pre-selected on load:

Languages

Source: Markdown

```
{% multiselect label='Languages' data='js::JavaScript, ts::TypeScript, py::Python, go::Go'
defaultValue='ts, py' %}
```

Source: Python

```
component('aardvark', 'multiselect',
          label='Languages',
          data='js::JavaScript, ts::TypeScript, py::Python, go::Go',
          defaultValue='ts, py')
```

Grouped options with dataJson

For grouped, disabled, or richer options pass a full JSON array through `dataJson` — a list of `{group, items}` objects. `dataJson` wins over `data`:

Tooling

Source: Markdown

```
{% multiselect label='Tooling' searchable=true
dataJson='[{"group":"Build","items":["Vite","esbuild","Webpack"]},{"group":"Test","items":["
Vitest","Jest","Playwright"]}]' %}
```

Source: Python

```
component('aardvark', 'multiselect',
          label='Tooling', searchable=True,
          dataJson='[{"group":"Build","items":["Vite","esbuild","Webpack"]},'
                   '{"group":"Test","items":["Vitest","Jest","Playwright"]}]')
```

Limiting and hiding picked options

`maxValues` caps how many can be chosen — once full, the rest are disabled. `hidePickedOptions` drops an option from the dropdown once it's selected:

Pick up to 2

Toppings

Source: Markdown

```
{% multiselect label='Pick up to 2' data='Red, Green, Blue, Yellow' maxValues=2 %}
{% multiselect label='Toppings' data='Cheese, Mushroom, Olive, Onion' hidePickedOptions=true
clearable=true %}
```

Source: Python

```
component('aardvark', 'multiselect', label='Pick up to 2', data='Red, Green, Blue, Yellow',
maxValues=2)
component('aardvark', 'multiselect', label='Toppings', data='Cheese, Mushroom, Olive,
Onion',
          hidePickedOptions=True, clearable=True)
```

With other components

A multiselect fits inside a card alongside other content — here a text lead-in above the field:

Select the frameworks your project uses.

Frameworks

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}Select the frameworks your project uses.{% endText %}

{% multiselect label='Frameworks' placeholder='Add a few' data='React, Vue, Svelte, Angular,
Solid' searchable=true clearable=true %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='Select the frameworks your project uses.',
size='sm', c='dimmed') +
    component('aardvark', 'multiselect',
              label='Frameworks', placeholder='Add a few',
              data='React, Vue, Svelte, Angular, Solid',
              searchable=True, clearable=True)
))
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The body is ignored — `multiselect` is a form control, not a container.

Attribute	Valid values	Description
`data`	Comma-separated list, or `value::label` pairs (double colon)	The options. A bare item is both value and label; `value::label` carries a separate stored value. A single `:` is left intact.
`dataJson`	JSON array string	A full options array — plain strings, `{value, label}` objects, `{group, items}` groups, and disabled flags. Wins over `data` when both are set.
`defaultValue`	Comma-separated list of values from `data`	The options selected on load.

`maxValues`	Integer	Cap the number of selections; further options are disabled once full.
`label`	String	The field label.
`placeholder`	String	Empty-state text inside the input.
`description`	String	Helper text below the label.
`error`	String	Validation message; also styles the field red.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Input size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`searchable`	`true`, `false` (default `false`)	Let the reader filter options by typing.
`clearable`	`true`, `false` (default `false`)	Show an × to clear all selections.
`hidePickedOptions`	`true`, `false` (default `false`)	Remove an option from the dropdown once it’s picked.
`disabled`	`true`, `false` (default `false`)	Disable the input.
`required`	`true`, `false` (default `false`)	Mark the field required.
`withAsterisk`	`true`, `false` (default `false`)	Show the required asterisk on the label.
`withCheckIcon`	`true`, `false` (default `true`)	Show the check mark on a selected option.
`checkIconPosition`	`left`, `right`	Which side the check mark sits on.
`withScrollArea`	`true`, `false` (default `true`)	Wrap a long dropdown in a scroll area.

`nothingFoundMessage`	String	Text shown when a search matches nothing.
`maxDropdownHeight`	Integer (px)	Cap the dropdown height.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="MultiSelect"]` — or through the Mantine Styles API classes (`.mantine-MultiSelect-root` and its inner parts):

```
/* Every rendered MultiSelect carries this island marker */
[data-aardvark-island="MultiSelect"] { }

/* Mantine Styles API class on the root element */
.mantine-MultiSelect-root { }
.mantine-MultiSelect-input { }
.mantine-MultiSelect-pill { }
.mantine-MultiSelect-label { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — here it is wired to `onchange` on the search box, so typing to filter logs the text you type to the console and alerts it — selecting items is React state and fires no DOM `change` event:

Stack

Source: Markdown

```
{% multiselect label='Stack' placeholder='Pick a few' data='React, Vue, Svelte, Angular,
Solid' searchable=true clearable=true attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'multiselect',
          label='Stack', placeholder='Pick a few',
          data='React, Vue, Svelte, Angular, Solid',
          searchable=True, clearable=True, attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
```

```
'''})
```

Pill

A built-in tag for a small rounded pill — the chip that MultiSelect and TagsInput render for each value. Use it on its own wherever you want a compact token. The label is the block body or a `text` param.

Use it as `{% pill %}…{% endPill %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'pill', …)`.

Basic pills

The label is the block body, or the `text` param for a self-closing tag:

react

v1.0

Source: Markdown

```
{% pill %}react{% endPill %}
{% pill text='v1.0' %}
```

Source: Python

```
component('aardvark', 'pill', children='react')
component('aardvark', 'pill', text='v1.0')
```

Variant, remove button, disabled

`variant` is `default` or `contrast`. `withRemoveButton` shows an × (visual only here — the real remove behavior lives inside MultiSelect and TagsInput). `disabled` dims the pill:

contrast

removable

disabled

Source: Markdown

```
{% pill variant='contrast' %}contrast{% endPill %}
{% pill withRemoveButton=true %}removable{% endPill %}
{% pill disabled=true %}disabled{% endPill %}
```

Source: Python

```
component('aardvark', 'pill', children='contrast', variant='contrast')
component('aardvark', 'pill', children='removable', withRemoveButton=True)
```

```
component('aardvark', 'pill', children='disabled', disabled=True)
```

Sizes and radius

`size` and `radius` both run `xs`–`xl`:

xs

md

xl

Source: Markdown

```
{% pill size='xs' %}xs{% endPill %}
{% pill size='md' %}md{% endPill %}
{% pill size='xl' radius='xl' %}xl{% endPill %}
```

Source: Python

```
component('aardvark', 'pill', children='xs', size='xs')
component('aardvark', 'pill', children='md', size='md')
component('aardvark', 'pill', children='xl', size='xl', radius='xl')
```

Building a row of pills in Python

Because the tag is also a Python callable, a bare `{% … %}` template block can loop and emit a row of pills — handy when the labels come from data rather than being typed out:

docs

markdown

mantine

static-site

Source: Python

```
{%
print(' '.join(
    component('aardvark', 'pill', children=tag)
    for tag in ['docs', 'markdown', 'mantine', 'static-site']
))
%}
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The label comes from the block body, or from `text` when self-closing.

Attribute	Valid values	Description
`text`	String	The label, when not using the block body. HTML-escaped.
`variant`	`default`, `contrast`	Pill style.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Pill size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`withRemoveButton`	`true`, `false` (default `false`)	Show an × remove button (visual only; no handler is wired in this standalone tag).
`disabled`	`true`, `false` (default `false`)	Render in a disabled state.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Pill"]` — or through the Mantine Styles API classes (`.mantine-Pill-root` and its inner parts):

```
/* Every rendered Pill carries this island marker */
[data-aardvark-island="Pill"] { }

/* Mantine Styles API class on the root element */
.mantine-Pill-root { }
.mantine-Pill-label { }
.mantine-Pill-remove { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

react

Source: Markdown

```
{% pill attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}react{% endPill %}
```

Source: Python

```
component('aardvark', 'pill', children='react', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

PillsInput

`PillsInput` is the input shell that holds a row of pills plus a text field — the composition primitive MultiSelect and TagsInput are built on top of. On its own it renders only the bordered, labelled container; it does not manage selection, add or remove pills, or filter options. Put Pills in the body to fill it. For a ready-made interactive multi-value field reach for MultiSelect or TagsInput; `PillsInput` is here for the rare case where you want the shell's look around your own static content.

Use it as `{% pillsinput %}…{% endPillsinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'pillsinput', …)`.

Basic shell

Put Pills in the body. The tag gives them a labelled, bordered shell. `description` adds helper text below the label:

Tags

docs

markdown

Source: Markdown

```
{% pillsinput label='Tags' description='A static demo shell' %}
{% pill withRemoveButton=true %}docs{% endPill %}
{% pill withRemoveButton=true %}markdown{% endPill %}
{% endPillsinput %}
```

Source: Python

```
component('aardvark', 'pillsinput',
          label='Tags', description='A static demo shell',
          children=(
              component('aardvark', 'pill', children='docs', withRemoveButton=True) +
              component('aardvark', 'pill', children='markdown', withRemoveButton=True)
          ))
```

Sizes, variants, and required

`size` runs `xs`–`xl`; `variant` is `default`, `filled`, or `unstyled`; `radius` follows the same `xs`–`xl` scale. `required` with `withAsterisk` marks the field:

Filters

active

Source: Markdown

```
{% pillsinput label='Filters' size='lg' variant='filled' radius='md' required=true
withAsterisk=true %}
{% pill %}active{% endPill %}
{% endPillsinput %}
```

Source: Python

```
component('aardvark', 'pillsinput',
          label='Filters', size='lg', variant='filled', radius='md',
          required=True, withAsterisk=True,
          children=component('aardvark', 'pill', children='active'))
```

Showing an error

`error` shows a validation message and styles the shell red:

Tags

draft

Source: Markdown

```
{% pillsinput label='Tags' error='Pick at least one tag' %}
{% pill %}draft{% endPill %}
{% endPillsinput %}
```

Source: Python

```
component('aardvark', 'pillsinput',
          label='Tags', error='Pick at least one tag',
          children=component('aardvark', 'pill', children='draft'))
```

With other components

The shell sits inside a card alongside other content — here a text lead-in above it:

Applied filters (read-only).

Filters

status: open

label: bug

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}Applied filters (read-only).{% endText %}

{% pillsinput label='Filters' %}
{% pill withRemoveButton=true %}status: open{% endPill %}
{% pill withRemoveButton=true %}label: bug{% endPill %}
{% endPillsinput %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='Applied filters (read-only).', size='sm',
c='dimmed') +
    component('aardvark', 'pillsinput', label='Filters', children=(
        component('aardvark', 'pill', children='status: open', withRemoveButton=True) +
        component('aardvark', 'pill', children='label: bug', withRemoveButton=True)
    ))
))
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The block body is the shell's content — typically a set of Pills.

Attribute	Valid values	Description
`label`	String	The field label.
`description`	String	Helper text below the label.
`error`	String	Validation message; also styles the shell red.

`size`	`xs`, `sm`, `md`, `lg`, `xl`	Input size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`disabled`	`true`, `false` (default `false`)	Render in a disabled state.
`required`	`true`, `false` (default `false`)	Mark the field required.
`withAsterisk`	`true`, `false` (default `false`)	Show the required asterisk on the label.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="PillsInput"]` — or through the Mantine Styles API classes (`.mantine-PillsInput-root` and its inner parts):

```
/* Every rendered PillsInput carries this island marker */
[data-aardvark-island="PillsInput"] { }

/* Mantine Styles API class on the root element */
.mantine-PillsInput-root { }
.mantine-PillsInput-input { }
.mantine-PillsInput-label { }
.mantine-PillsInput-wrapper { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Tags

Source: Markdown

```
{% pillsinput label='Tags' description='A static demo shell' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'pillsinput',
          label='Tags', description='A static demo shell', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Select

A built-in tag for a single-select dropdown — the reader picks one option from a list. It hydrates into a fully interactive Mantine input in the browser, with no JavaScript to write. Give it a `label` and an options list through `data`, where the simplest form is a comma-separated list and `value::label` pairs let a stored value carry a friendly label.

Use it as `{% select %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'select', …)`.

Basic select

A `label`, a `placeholder`, and a comma-separated `data` list:

Framework

Source: Markdown

```
{% select label='Framework' placeholder='Pick one' data='React, Vue, Svelte, Angular' %}
```

Source: Python

```
component('aardvark', 'select',
          label='Framework', placeholder='Pick one',
          data='React, Vue, Svelte, Angular')
```

value::label pairs, searchable, clearable

`value::label` pairs (a double colon) let a short stored value carry a display label. `searchable` lets the reader filter by typing; `clearable` shows an × to reset:

Country

Source: Markdown

```
{% select label='Country' data='us::United States, ca::Canada, mx::Mexico' searchable=true
clearable=true %}
```

Source: Python

```
component('aardvark', 'select',
          label='Country',
          data='us::United States, ca::Canada, mx::Mexico',
          searchable=True, clearable=True)
```

Grouped options with dataJson

For grouped, disabled, or otherwise richer options, pass a full JSON array through `dataJson` — a list of `{group, items}` objects. `dataJson` wins over `data` when both are set:

Language

Source: Markdown

```
{% select label='Language' searchable=true
dataJson='[{"group":"Frontend","items":["JavaScript","TypeScript"]},{"group":"Backend","item
s":["Python","Go","Rust"]}]' %}
```

Source: Python

```
component('aardvark', 'select',
          label='Language', searchable=True,
          dataJson='[{"group":"Frontend","items":["JavaScript","TypeScript"]},'
                   '{"group":"Backend","items":["Python","Go","Rust"]}]')
```

Sizes and variants

`size` runs `xs`–`xl`; `variant` is `default`, `filled`, or `unstyled`; `radius` follows the same `xs`–`xl` scale. `defaultValue` selects an option on load:

Small

Filled

Source: Markdown

```
{% select label='Small' data='S, M, L' size='xs' defaultValue='M' %}
{% select label='Filled' data='One, Two, Three' variant='filled' radius='xl' %}
```

Source: Python

```
component('aardvark', 'select', label='Small', data='S, M, L', size='xs', defaultValue='M')
component('aardvark', 'select', label='Filled', data='One, Two, Three', variant='filled',
radius='xl')
```

Required, with description and error

`description` adds helper text below the label; `error` shows a validation message; `required` with `withAsterisk` marks the field and shows the asterisk:

Plan

Source: Markdown

```
{% select label='Plan' data='Free, Pro, Enterprise' description='Choose a billing tier'
required=true withAsterisk=true %}
```

Source: Python

```
component('aardvark', 'select',
          label='Plan', data='Free, Pro, Enterprise',
          description='Choose a billing tier',
          required=True, withAsterisk=True)
```

With other components

A select sits naturally inside a card alongside other content — here a short text lead-in above the field:

Set your preferred deployment region.

Region

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}Set your preferred deployment region.{% endText %}

{% select label='Region' data='us-east::US East, us-west::US West, eu::Europe, ap::Asia
Pacific' searchable=true clearable=true %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='Set your preferred deployment region.',
size='sm', c='dimmed') +
    component('aardvark', 'select',
              label='Region',
              data='us-east::US East, us-west::US West, eu::Europe, ap::Asia Pacific',
              searchable=True, clearable=True)
))
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The body is ignored — `select` is a form control, not a container.

Attribute	Valid values	Description
`data`	Comma-separated list, or `value::label` pairs (double colon)	The options. A bare item is both value and label; `value::label` carries a separate stored value. A single `:` is left intact (so URLs and times pass through).
`dataJson`	JSON array string	A full options array — plain strings, `{value, label}` objects, `{group, items}` groups, and `{value, label, disabled}` flags. Wins over `data` when both are set.
`label`	String	The field label.
`placeholder`	String	Empty-state text inside the input.
`description`	String	Helper text below the label.
`error`	String	Validation message; also styles the field red.
`defaultValue`	A value from `data`	The option selected on load.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Input size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`searchable`	`true`, `false` (default `false`)	Let the reader filter options by typing.
`clearable`	`true`, `false` (default `false`)	Show an × to clear the selection.
`disabled`	`true`, `false` (default `false`)	Disable the input.
`required`	`true`, `false` (default `false`)	Mark the field required.
`withAsterisk`	`true`, `false` (default `false`)	Show the required asterisk on the label.

`allowDeselect`	`true`, `false` (default `true`)	Click the selected option again to clear it.
`withCheckIcon`	`true`, `false` (default `true`)	Show the check mark on the selected option.
`checkIconPosition`	`left`, `right`	Which side the check mark sits on.
`withScrollArea`	`true`, `false` (default `true`)	Wrap a long dropdown in a scroll area.
`nothingFoundMessage`	String	Text shown when a search matches nothing.
`maxDropdownHeight`	Integer (px)	Cap the dropdown height.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Select"]` — or through the Mantine Styles API classes (`.mantine-Select-root` and its inner parts):

```
/* Every rendered Select carries this island marker */
[data-aardvark-island="Select"] { }

/* Mantine Styles API class on the root element */
.mantine-Select-root { }
.mantine-Select-input { }
.mantine-Select-label { }
.mantine-Select-wrapper { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered element. A read-only combobox manages its own click and selection (React state — there's no native `change`/`click` to hook from an inline handler), so `attr` is best here for a static `data-*` hook, an `id`, or ARIA. It lands on the rendered input, where scripts, tests, or DevTools can find it:

Framework

Source: Markdown

```
{% select label='Framework' placeholder='Pick one' data='React, Vue, Svelte, Angular'
attr={'data-testid': 'framework-select'} %}
```

Source: Python

```
component('aardvark', 'select',
          label='Framework', placeholder='Pick one',
          data='React, Vue, Svelte, Angular',
          attr={'data-testid': 'framework-select'})
```

TagsInput

A built-in tag for a free-text tags field: type a value, press Enter, and it becomes a removable Pill. Unlike MultiSelect the tags are free text — `data` is only an optional set of autocomplete suggestions, not a fixed option list. It hydrates into a fully interactive Mantine input in the browser.

Use it as `{% tagsinput %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'tagsinput', …)`.

Basic tags input

A `label`, a `placeholder`, and a `defaultValue` of pre-filled tags (a comma-separated list):

Tags

Source: Markdown

```
{% tagsinput label='Tags' placeholder='Type and press Enter' defaultValue='docs, markdown'
%}
```

Source: Python

```
component('aardvark', 'tagsinput',
          label='Tags', placeholder='Type and press Enter',
          defaultValue='docs, markdown')
```

With suggestions

`data` gives autocomplete suggestions as the reader types; they are plain strings, since the typed value is free text. `clearable` shows an × to drop all tags:

Skills

Source: Markdown

```
{% tagsinput label='Skills' data='Python, JavaScript, Go, Rust' clearable=true %}
```

Source: Python

```
component('aardvark', 'tagsinput',
          label='Skills', data='Python, JavaScript, Go, Rust',
          clearable=True)
```

Limiting and splitting

`maxTags` caps the count; `splitChars` adds separators that commit a tag (the default is Enter only). Here a comma also commits, up to three tags:

Up to 3, comma also commits

Source: Markdown

```
{% tagsinput label='Up to 3, comma also commits' maxTags=3 splitChars=',' %}
```

Source: Python

```
component('aardvark', 'tagsinput',
          label='Up to 3, comma also commits',
          maxTags=3, splitChars=',')
```

Allowing duplicates

By default a tag can appear only once. `allowDuplicates` lets the same tag be added again; `acceptValueOnBlur` (on by default) commits a pending tag when the field loses focus:

Log entries

Source: Markdown

```
{% tagsinput label='Log entries' defaultValue='start' allowDuplicates=true %}
```

Source: Python

```
component('aardvark', 'tagsinput',
          label='Log entries', defaultValue='start',
          allowDuplicates=True)
```

With other components

A tags input fits inside a card alongside other content — here a text lead-in above the field:

Tag this article for search.

Tags

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}Tag this article for search.{% endText %}

{% tagsinput label='Tags' placeholder='Type and press Enter' data='docs, guide, reference,
tutorial' clearable=true %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='Tag this article for search.', size='sm',
c='dimmed') +
    component('aardvark', 'tagsinput',
              label='Tags', placeholder='Type and press Enter',
              data='docs, guide, reference, tutorial', clearable=True)
))
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The body is ignored — `tagsinput` is a form control, not a container.

Attribute	Valid values	Description
`data`	Comma-separated list	Optional autocomplete suggestions — plain strings, since tags are free text.
`dataJson`	JSON array string	A full suggestions array, including `{group, items}` groups. Wins over `data` when both are set.
`defaultValue`	Comma-separated list	The tags shown on load.
`splitChars`	Comma-separated characters	Separators that commit a tag (in addition to Enter). For example `,` or `,|` for several.
`maxTags`	Integer	Cap the number of tags.
`label`	String	The field label.
`placeholder`	String	Empty-state text inside the input.
`description`	String	Helper text below the label.
`error`	String	Validation message; also styles the field red.

`size`	`xs`, `sm`, `md`, `lg`, `xl`	Input size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`searchable`	`true`, `false` (default `false`)	Filter the suggestion list by typing.
`clearable`	`true`, `false` (default `false`)	Show an × to clear all tags.
`acceptValueOnBlur`	`true`, `false` (default `true`)	Commit the pending value when the field loses focus.
`allowDuplicates`	`true`, `false` (default `false`)	Allow the same tag more than once.
`disabled`	`true`, `false` (default `false`)	Disable the input.
`required`	`true`, `false` (default `false`)	Mark the field required.
`withAsterisk`	`true`, `false` (default `false`)	Show the required asterisk on the label.
`maxDropdownHeight`	Integer (px)	Cap the suggestion dropdown height.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="TagsInput"]` — or through the Mantine Styles API classes (`.mantine-TagsInput-root` and its inner parts):

```
/* Every rendered TagsInput carries this island marker */
[data-aardvark-island="TagsInput"] { }

/* Mantine Styles API class on the root element */
.mantine-TagsInput-root { }
.mantine-TagsInput-input { }
.mantine-TagsInput-pill { }
.mantine-TagsInput-label { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — here it is wired to `onchange` on the search box, so typing to filter logs the text you type to the console and alerts it — selecting items is React state and fires no DOM `change` event:

Tags

Source: Markdown

```
{% tagsinput label='Tags' placeholder='Type and press Enter' defaultValue='docs, markdown'
attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'tagsinput',
          label='Tags', placeholder='Type and press Enter',
          defaultValue='docs, markdown', attr={'onchange': '''
const value = this.value;
console.log('attr demo value:', value);
alert(value);
'''})
```

TreeSelect

A built-in tag for selecting one value from a hierarchy — a dropdown whose options are a collapsible tree. Click a folder to expand it; click a leaf to select it. Set `searchable` to filter the tree as you type. It hydrates into a fully interactive island just like the other select tags.

Use it as `{% treeselect %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'treeselect', …)`.

Basic tree select

The hierarchy comes through `data` — a JSON array of nested nodes, each an object with a `value`, a `label`, and an optional `children` array of the same shape. Only leaf nodes (those without `children`) resolve to a selection; clicking a branch just expands or collapses it. `clearable` shows an × to reset:

Pick a file

Source: Markdown

```
{% treeselect label='Pick a file' placeholder='Browse…' clearable=true data='[
  {"value":"src","label":"src","children":[
    {"value":"src/app.py","label":"app.py"},
    {"value":"src/utils","label":"utils","children":[
      {"value":"src/utils/text.py","label":"text.py"}
    ]}
  ]},
  {"value":"README.md","label":"README.md"}
]' %}
```

Source: Python

```
component('aardvark', 'treeselect',
          label='Pick a file', placeholder='Browse…', clearable=True,
          data='''[
  {"value":"src","label":"src","children":[
    {"value":"src/app.py","label":"app.py"},
    {"value":"src/utils","label":"utils","children":[
      {"value":"src/utils/text.py","label":"text.py"}
    ]}
  ]},
  {"value":"README.md","label":"README.md"}
]''')
```

Sizes, variants, description, and search

`size` runs `xs`–`xl`; `variant` is `default`, `filled`, or `unstyled`. `description` adds helper text below the label, and `error` shows a validation message. `searchable` adds a search field that filters the tree as you type:

Category

Source: Markdown

```
{% treeselect label='Category' description='Pick the deepest matching node' size='md'
variant='filled' radius='md' searchable=true data='[
  {"value":"hardware","label":"Hardware","children":[
    {"value":"hardware/laptops","label":"Laptops"},
    {"value":"hardware/phones","label":"Phones"}
  ]},
  {"value":"software","label":"Software","children":[
    {"value":"software/os","label":"Operating systems"}
  ]}
]' %}
```

Source: Python

```
component('aardvark', 'treeselect',
          label='Category', description='Pick the deepest matching node',
          size='md', variant='filled', radius='md', searchable=True,
          data='''[
  {"value":"hardware","label":"Hardware","children":[
    {"value":"hardware/laptops","label":"Laptops"},
    {"value":"hardware/phones","label":"Phones"}
  ]},
  {"value":"software","label":"Software","children":[
    {"value":"software/os","label":"Operating systems"}
  ]}
]''')
```

Data shape

Each node is an object:

```
{
  "value": "unique-id",
  "label": "Display text",
  "children": [ /* more nodes, optional */ ]
}
```

`value` must be unique across the whole tree — it's what gets selected; `label` is what the reader sees. A node with no `children` is a selectable leaf. Unlike the flat select family, `treeselect` takes only this nested JSON tree, not a comma-separated `data` list.

With other components

A tree select fits inside a card alongside other content — here a text lead-in above the field:

Choose where this document lives.

Location

Source: Markdown

```
{% card %}
{% text size='sm' c='dimmed' %}Choose where this document lives.{% endText %}

{% treeselect label='Location' placeholder='Browse…' clearable=true data='[
  {"value":"docs","label":"docs","children":[
    {"value":"docs/guide","label":"guide"},
    {"value":"docs/reference","label":"reference"}
  ]},
  {"value":"blog","label":"blog"}
]' %}
{% endCard %}
```

Source: Python

```
tree = '''[
  {"value":"docs","label":"docs","children":[
    {"value":"docs/guide","label":"guide"},
    {"value":"docs/reference","label":"reference"}
  ]},
  {"value":"blog","label":"blog"}
]'''
component('aardvark', 'card', children=(
    component('aardvark', 'text', children='Choose where this document lives.', size='sm',
c='dimmed') +
    component('aardvark', 'treeselect',
              label='Location', placeholder='Browse…', clearable=True, data=tree)
))
```

Attributes

Every attribute is optional; omit one to take its Mantine default. The body is ignored — `treeselect` is a form control, not a container.

Attribute	Valid values	Description

`data`	JSON array string	The hierarchy — a JSON array of nested `{value, label, children}` nodes. A node without `children` is a selectable leaf.
`label`	String	The field label.
`placeholder`	String	Empty-state text on the trigger.
`description`	String	Helper text below the label.
`error`	String	Validation message; also styles the field red.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Input size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius.
`variant`	`default`, `filled`, `unstyled`	Input style.
`clearable`	`true`, `false` (default `false`)	Show an × to clear the selection.
`searchable`	`true`, `false` (default `false`)	Add a search field that filters the tree as you type.
`disabled`	`true`, `false` (default `false`)	Disable the input.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="TreeSelect"]` — or through the Mantine Styles API classes (`.mantine-TreeSelect-root` and its inner parts):

```
/* Every rendered TreeSelect carries this island marker */
[data-aardvark-island="TreeSelect"] { }

/* Mantine Styles API class on the root element */
.mantine-TreeSelect-root { }
.mantine-TreeSelect-input { }
.mantine-TreeSelect-label { }
.mantine-TreeSelect-wrapper { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered element. A read-only combobox manages its own click and selection (React state — there's no native `change`/`click` to hook from an inline handler), so `attr` is best here for a static `data-*` hook, an `id`, or ARIA. It lands on the rendered input, where scripts, tests, or DevTools can find it:

Pick a file

Source: Markdown

```
{% treeselect label='Pick a file' placeholder='Browse…' clearable=true data='[
  {"value":"src","label":"src","children":[
    {"value":"src/app.py","label":"app.py"}
  ]},
  {"value":"README.md","label":"README.md"}
]' attr={'data-testid': 'file-treeselect'} %}
```

Source: Python

```
component('aardvark', 'treeselect',
          label='Pick a file', placeholder='Browse…', clearable=True,
          data='''[
  {"value":"src","label":"src","children":[
    {"value":"src/app.py","label":"app.py"}
  ]},
  {"value":"README.md","label":"README.md"}
]''',
          attr={'data-testid': 'file-treeselect'})
```

Buttons

Aardvark's **built-in button tags** cover the whole family — buttons, icon buttons, copy and file pickers, and the unstyled primitive — each a single tag with no setup. For the underlying components and every prop they accept, see Mantine's button components.

- Button — the standard styled button or link: variant, color, size, gradient, sections, link target/rel/download, spacing, `id`, `onclick`.
- ActionIcon — an icon-only button. The `icon` is read exactly like the `{% icon %}` tag (Tabler, Font Awesome, image, emoji, or inline SVG).
- CloseButton — a dismiss button, with an optional custom icon.
- CopyButton — a button that copies a value to the clipboard and confirms the copy.
- FileButton — a button that opens the native file picker.
- UnstyledButton — a clean, accessible button (or link) with no styles, for you to style yourself.

ActionIcon

`{% actionicon %}` is a **built-in** tag for an **icon-only button** — a compact, square button whose whole content is a single icon. It renders a Mantine ActionIcon, so it carries that component's full surface. The `icon` is read exactly like the `{% icon %}` tag, and an icon-only button needs a `label` for its accessible name.

Use it as `{% actionicon %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'actionicon', …)`.

Like

Source: Markdown

```
{% actionicon icon='heart' variant='filled' color='red' label='Like' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='heart', variant='filled', color='red',
label='Like')
```

The icon

The `icon` is any `{% icon %}` spec:

- a **Tabler name** — a bare lowercase name like `settings` or `brand-github`;
- a **Font Awesome class** — `fa-solid fa-gear`;
- a path/URL to an **image** — `/icons/logo.svg`;
- an **emoji** — .

Set `filled` to use the filled Tabler style for a bare Tabler name.

Settings

GitHub

Settings

Search

Source: Markdown

```
{% actionicon icon='settings' label='Settings' variant='default' %}
{% actionicon icon='brand-github' label='GitHub' variant='default' %}
{% actionicon icon='fa-solid fa-gear' label='Settings' variant='default' %}
{% actionicon icon='' label='Search' variant='default' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='settings', label='Settings', variant='default')
component('aardvark', 'actionicon', icon='brand-github', label='GitHub', variant='default')
component('aardvark', 'actionicon', icon='fa-solid fa-gear', label='Settings',
variant='default')
component('aardvark', 'actionicon', icon='', label='Search', variant='default')
```

Accessible name

An icon-only button has no visible text, so screen readers can't name it. Always pass a `label` — it becomes the button's `aria-label`.

Delete item

Source: Markdown

```
{% actionicon icon='trash' color='red' label='Delete item' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='trash', color='red', label='Delete item')
```

Variant

`variant` is one of `filled`, `light` (default), `outline`, `subtle`, `transparent`, `white`, `default`, or `gradient`.

filled

light

outline

subtle

default

transparent

Source: Markdown

```
{% actionicon icon='star' variant='filled' label='filled' %}
{% actionicon icon='star' variant='light' label='light' %}
{% actionicon icon='star' variant='outline' label='outline' %}
{% actionicon icon='star' variant='subtle' label='subtle' %}
{% actionicon icon='star' variant='default' label='default' %}
{% actionicon icon='star' variant='transparent' label='transparent' %}
```

Source: Python

```
for v in ('filled', 'light', 'outline', 'subtle', 'default', 'transparent'):
    component('aardvark', 'actionicon', icon='star', variant=v, label=v)
```

Gradient

`variant='gradient'` takes `gradientFrom`, `gradientTo`, and `gradientDeg` (degrees).

Sunset

Indigo to cyan

Source: Markdown

```
{% actionicon icon='bolt' variant='gradient' gradientFrom='orange' gradientTo='red'
gradientDeg=45 label='Sunset' %}
{% actionicon icon='bolt' variant='gradient' gradientFrom='indigo' gradientTo='cyan'
gradientDeg=90 label='Indigo to cyan' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='bolt', variant='gradient',
          gradientFrom='orange', gradientTo='red', gradientDeg=45, label='Sunset')
component('aardvark', 'actionicon', icon='bolt', variant='gradient',
          gradientFrom='indigo', gradientTo='cyan', gradientDeg=90, label='Indigo to cyan')
```

Color, size, and radius

`color` takes any theme color or a CSS color. `size` accepts the Mantine tokens (`xs`–`xl`); `radius` accepts `xs`–`xl` or any CSS value. `autoContrast` auto-picks a glyph color that meets contrast against the button background.

grape

teal

lg + xl radius

xl

autoContrast

Source: Markdown

```
{% actionicon icon='heart' color='grape' label='grape' %}
{% actionicon icon='heart' color='teal' label='teal' %}
{% actionicon icon='heart' size='lg' radius='xl' label='lg + xl radius' %}
{% actionicon icon='heart' size='xl' label='xl' %}
{% actionicon icon='heart' color='yellow' autoContrast label='autoContrast' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='heart', color='grape', label='grape')
component('aardvark', 'actionicon', icon='heart', color='teal', label='teal')
```

```
component('aardvark', 'actionicon', icon='heart', size='lg', radius='xl', label='lg + xl
radius')
component('aardvark', 'actionicon', icon='heart', size='xl', label='xl')
component('aardvark', 'actionicon', icon='heart', color='yellow', autoContrast=True,
label='autoContrast')
```

Loading and disabled

`loading` shows a spinner (and blocks clicks); `disabled` renders the button disabled.

Saving

Disabled

Source: Markdown

```
{% actionicon icon='check' loading=true label='Saving' %}
{% actionicon icon='check' disabled=true label='Disabled' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='check', loading=True, label='Saving')
component('aardvark', 'actionicon', icon='check', disabled=True, label='Disabled')
```

Link mode

Set `url` and the button renders as a link (`<a href>`): it navigates on click and works without JavaScript. `target`, `rel`, and `download` pass through, exactly as on `{% button %}`.

GitHub

Source: Markdown

```
{% actionicon icon='brand-github' variant='default' url='https://github.com' target='_blank'
rel='noopener noreferrer' label='GitHub' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='brand-github', variant='default',
          url='https://github.com', target='_blank', rel='noopener noreferrer',
label='GitHub')
```

With other components

Icon-only buttons are the natural toolbar control — group several in a `{% group %}` next to a heading:

Edit

Duplicate

Delete

Source: Markdown

```
{% group gap='xs' %}
{% actionicon icon='edit' variant='subtle' label='Edit' %}
{% actionicon icon='copy' variant='subtle' label='Duplicate' %}
{% actionicon icon='trash' variant='subtle' color='red' label='Delete' %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='edit', variant='subtle', label='Edit')
component('aardvark', 'actionicon', icon='copy', variant='subtle', label='Duplicate')
component('aardvark', 'actionicon', icon='trash', variant='subtle', color='red',
label='Delete')
```

Attributes

Omit any attribute to take its Mantine default. Bare boolean flags (`filled`, `loading`, `disabled`, `autoContrast`) set the option to `True`; in Python pass `=True`.

Attribute	Valid values	Description
`icon`	Tabler name / Font Awesome class / image path / emoji / inline `<svg>`	The icon spec, read like `{% icon %}`.
`filled`	bool flag	Use the filled Tabler style for a bare Tabler name.
`label`	string	Accessible name (`aria-label`) — always set this.
`variant`	`filled`, `light` (default), `outline`, `subtle`, `transparent`, `white`, `default`, `gradient`	Visual style.
`color`	theme color name or CSS color	Button color.
`size`	`xs`–`xl`	Button size.

`radius`	`xs`–`xl` or any CSS value	Corner radius.
`loading`	bool flag	Show a spinner and block clicks.
`disabled`	bool flag	Render disabled.
`autoContrast`	bool flag	Auto-pick a glyph color that meets contrast.
`url`	relative path or `http(s)://` URL	Render as a link (`<a href>`). `javascript:`/`data:`/`vbscript:`/`file:`/`blob:` schemes are rejected at build time.
`target`	e.g. `_blank`	Where to open the link (link mode).
`rel`	e.g. `noopener noreferrer`	Link relationship (link mode).
`download`	filename string	Suggest a filename for the linked file (link mode).
`id`	string	HTML `id` on the rendered button (or `<a>`).
`onclick`	JS expression string	JavaScript run on click. In Python pass `attr={'onclick': '…'}`.
`gradientFrom`	color	Start color (`variant='gradient'`).
`gradientTo`	color	End color (`variant='gradient'`).
`gradientDeg`	number (degrees)	Angle (`variant='gradient'`).
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	size token or CSS value	Margin (Mantine spacing system).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	size token or CSS value	Padding (Mantine spacing system).

CSS Selectors

Each action icon mounts inside an island wrapper carrying `data-aardvark-island="ActionIcon"`; Mantine's Styles API exposes the button root, the icon slot, and the loading overlay.

```
[data-aardvark-island="ActionIcon"]  /* the island wrapper */
.mantine-ActionIcon-root             /* the <button> */
.mantine-ActionIcon-icon             /* the icon slot */
```

```
.mantine-ActionIcon-loader           /* the loading spinner (loading=true) */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered button. The `onclick` shortcut is the common case: set `onclick` to a JavaScript expression and it runs when the button is clicked. In Python, `onclick` rides this same channel — pass `attr={'onclick': '…'}`.

Ring

Source: Markdown

```
{% actionicon icon='bell' onclick="() => alert('Ding!'))()" label='Ring' %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='bell', label='Ring',
          attr={'onclick': "() => alert('Ding!'))()"})
```

For any other attribute — `data-*`, ARIA, or a full multi-line handler — pass the `attr={…}` dict directly:

Like

Source: Markdown

```
{% actionicon icon='heart' variant='filled' color='red' label='Like' attr={'onclick': '''
const value = this.getAttribute('aria-label');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'actionicon', icon='heart', variant='filled', color='red',
label='Like',
          attr={'onclick': '''
const value = this.getAttribute('aria-label');
console.log('attr demo value:', value);
alert(value);
'''})
```

(See the `{% button %}` page's note about `onclick` as a stored-XSS surface on multi-author sites, and the site-wide `attrPolicy`.)

Button

`{% button %}` is a **built-in** tag for buttons and button-styled links — every Mantine Button capability, exposed as a single tag with no setup. The header's top-bar call-to-action buttons (`topButtons` in `aardvark.config.yaml`) use the same tag.

Use it as `{% button %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'button', …)`.

Get started

Source: Markdown

```
{% button text='Get started' url='#' %}
```

Source: Python

```
component('aardvark', 'button', text='Get started', url='#')
```

Label

Give the label inline with `text`, or as the block body (handy when the label wraps other markup). In Python, pass the body as `children`.

Inline label

Block label

Source: Markdown

```
{% button text='Inline label' %}
{% button %}Block label{% endButton %}
```

Source: Python

```
component('aardvark', 'button', text='Inline label')
component('aardvark', 'button', children='Block label')
```

Linking

Set `url` and the button renders as a link (`<a href>`): it navigates on click and works even without JavaScript. A scoped style keeps the link looking like a button rather than inheriting the theme's prose-link color and underline.

Browse components

Plain button

Source: Markdown

```
{% button text='Browse components' url='/components/' %}
{% button text='Plain button' %}
```

Source: Python

```
component('aardvark', 'button', text='Browse components', url='/components/')
component('aardvark', 'button', text='Plain button')
```

Open in a new tab / download a file

When `url` is set, link-only attributes pass through: `target`, `rel`, `download`. Pair `target='_blank'` with `rel='noopener noreferrer'` for security.

Mantine docs

Download report

Source: Markdown

```
{% button text='Mantine docs' url='https://mantine.dev' target='_blank' rel='noopener
noreferrer' %}
{% button text='Download report' url='/report.pdf' download='report.pdf' variant='outline'
%}
```

Source: Python

```
component('aardvark', 'button', text='Mantine docs', url='https://mantine.dev',
          target='_blank', rel='noopener noreferrer')
component('aardvark', 'button', text='Download report', url='/report.pdf',
          download='report.pdf', variant='outline')
```

Variant

`variant` is one of `filled` (default), `outline`, `light`, `subtle`, `default`, `transparent`, `white`, or `gradient`.

filled

outline

light

subtle

default

Source: Markdown

```
{% button text='filled' variant='filled' %}
{% button text='outline' variant='outline' %}
{% button text='light' variant='light' %}
{% button text='subtle' variant='subtle' %}
{% button text='default' variant='default' %}
```

Source: Python

```
for v in ('filled', 'outline', 'light', 'subtle', 'default'):
    component('aardvark', 'button', text=v, variant=v)
```

Gradient

`variant='gradient'` takes `gradientFrom`, `gradientTo`, and `gradientDeg` (a number, in degrees). They compose into a single Mantine gradient. Supply all three together — if you omit any, Mantine fills in its default for the missing field, usually not what you intended.

Sunset

Indigo → cyan

Source: Markdown

```
{% button text='Sunset' variant='gradient' gradientFrom='orange' gradientTo='red'
gradientDeg=45 %}
{% button text='Indigo → cyan' variant='gradient' gradientFrom='indigo' gradientTo='cyan'
gradientDeg=90 %}
```

Source: Python

```
component('aardvark', 'button', text='Sunset', variant='gradient',
          gradientFrom='orange', gradientTo='red', gradientDeg=45)
component('aardvark', 'button', text='Indigo → cyan', variant='gradient',
          gradientFrom='indigo', gradientTo='cyan', gradientDeg=90)
```

Color

`color` takes any theme color (a brand color from your theme's `theme.scss`, like `primary` or `secondary`) or a CSS color. It defaults to the theme's primary.

grape

teal

custom hex

Source: Markdown

```
{% button text='grape' color='grape' %}
```

```
{% button text='teal' variant='outline' color='teal' %}
{% button text='custom hex' color='#e8590c' %}
```

Source: Python

```
component('aardvark', 'button', text='grape', color='grape')
component('aardvark', 'button', text='teal', variant='outline', color='teal')
component('aardvark', 'button', text='custom hex', color='#e8590c')
```

`autoContrast` picks a label color (white or black) that meets contrast against the button's background — useful with `filled` on light or non-Mantine colors.

autoContrast

no autoContrast

Source: Markdown

```
{% button text='autoContrast' color='yellow' autoContrast %}
{% button text='no autoContrast' color='yellow' %}
```

Source: Python

```
component('aardvark', 'button', text='autoContrast', color='yellow', autoContrast=True)
component('aardvark', 'button', text='no autoContrast', color='yellow')
```

Size, radius, and full-width

`size` accepts the Mantine size tokens (`xs`–`xl`) plus the compact variants (`compact-xs`–`compact-xl`). `radius` accepts `xs`–`xl` or any CSS value. `fullWidth` stretches the button to its container's width.

compact-md

md (default)

lg + xl radius

Source: Markdown

```
{% button text='compact-md' size='compact-md' %}
{% button text='md (default)' %}
{% button text='lg + xl radius' size='lg' radius='xl' %}
```

Source: Python

```
component('aardvark', 'button', text='compact-md', size='compact-md')
component('aardvark', 'button', text='md (default)')
component('aardvark', 'button', text='lg + xl radius', size='lg', radius='xl')
```

Sections

`leftSection` and `rightSection` render content beside the label — a small string is the simplest case (emoji, arrow, short text). The Button's flex layout gives them the right spacing automatically.

Continue

New

Source: Markdown

```
{% button text='Continue' rightSection='→' %}
{% button text='New' leftSection='' variant='outline' %}
```

Source: Python

```
component('aardvark', 'button', text='Continue', rightSection='→')
component('aardvark', 'button', text='New', leftSection='', variant='outline')
```

When `fullWidth` is on, `justify` controls how the sections and label distribute horizontally (any CSS `justify-content` value):

Settings

Source: Markdown

```
{% button text='Settings' leftSection='' rightSection='›' fullWidth justify='space-between'
%}
```

Source: Python

```
component('aardvark', 'button', text='Settings', leftSection='', rightSection='›',
          fullWidth=True, justify='space-between')
```

Form buttons

Inside a `<form>`, `type='submit'` makes the button submit the form (the HTML default; `type='reset'` and `type='button'` are also accepted).

Submit

Reset

Source: Markdown

```
{% button text='Submit' type='submit' %}
{% button text='Reset' type='reset' variant='outline' %}
```

Source: Python

```
component('aardvark', 'button', text='Submit', type='submit')
component('aardvark', 'button', text='Reset', type='reset', variant='outline')
```

Disabled

Disabled

Disabled outline

Source: Markdown

```
{% button text='Disabled' disabled %}
{% button text='Disabled outline' variant='outline' disabled %}
```

Source: Python

```
component('aardvark', 'button', text='Disabled', disabled=True)
component('aardvark', 'button', text='Disabled outline', variant='outline', disabled=True)
```

Selectors (`id`)

Set `id` to give CSS and JS something to target. It lands on the rendered button (or `<a>` in link mode), so `#cta` works in stylesheets and `document.getElementById('cta')` in scripts.

Pick me

Source: Markdown

```
{% button text='Pick me' id='cta' %}
```

Source: Python

```
component('aardvark', 'button', text='Pick me', id='cta')
```

Spacing & sizing

The Button accepts Mantine's spacing system — margin (`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`) and padding (`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`) with the same size tokens as everywhere else (`xs`, `sm`, `md`, `lg`, `xl`) or any CSS value. `bg`/`c` shortcut background and text color, and `w`/`h`/`miw`/`mih`/`maw`/`mah` size the button.

Wide

With air

Painted

Source: Markdown

```
{% button text='Wide' w='240' my='sm' %}
{% button text='With air' m='md' px='xl' %}
{% button text='Painted' bg='grape.6' c='white' %}
```

Source: Python

```
component('aardvark', 'button', text='Wide', w='240', my='sm')
component('aardvark', 'button', text='With air', m='md', px='xl')
component('aardvark', 'button', text='Painted', bg='grape.6', c='white')
```

With other components

A button sits naturally inside other primitives — here in a card, with a `{% group %}` to lay a pair side by side:

Upgrade your plan

Unlock private repos, custom domains, and priority support.

Upgrade

Compare plans

Source: Markdown

```
{% card title='Upgrade your plan' %}
Unlock private repos, custom domains, and priority support.

{% group %}
{% button text='Upgrade' variant='gradient' gradientFrom='grape' gradientTo='pink'
gradientDeg=135 %}
{% button text='Compare plans' variant='subtle' url='/components/' %}
{% endGroup %}
{% endCard %}
```

Source: Python

```
component('aardvark', 'button', text='Upgrade', variant='gradient',
          gradientFrom='grape', gradientTo='pink', gradientDeg=135)
component('aardvark', 'button', text='Compare plans', variant='subtle', url='/components/')
```

In the header

The site's top-bar call-to-action buttons are the same tag. Configure them with `topButtons` in `aardvark.config.yaml` — each entry accepts every field below. The default is a **Login** (outline) + **Sign up** (grape→pink gradient at 135°) pair; set `topButtons: []` to hide them.

Attributes

Omit any attribute to take its Mantine default. Bare boolean flags (`fullWidth`, `disabled`, `autoContrast`) set the option to `True`; in Python pass `=True`.

Label

Attribute	Valid values	Description
`text`	string	Label, when not using the block body.

Link mode (`url` set → `<a href>`)

Attribute	Valid values	Description
`url`	relative path or `http(s)://` URL	Navigate here on click; works without JS. `javascript:`/`data:`/`vbscript:`/`file:`/`blob:` schemes are rejected at build time.
`target`	e.g. `_blank`	Where to open the link (`_blank` for a new tab).
`rel`	e.g. `noopener noreferrer`	Link relationship; pair with `target='_blank'`.
`download`	filename string	Suggest a filename for the linked file.

State & behavior

Attribute	Valid values	Description
`variant`	`filled` (default), `outline`, `light`, `subtle`, `default`, `transparent`, `white`, `gradient`	Visual style.
`color`	theme color name or CSS color	Button color. Defaults to the theme primary.
`size`	`xs`–`xl`, `compact-xs`–`compact-xl`	Button size.
`radius`	`xs`–`xl` or any CSS value	Corner radius.
`fullWidth`	bool flag	Stretch to the container's width.
`disabled`	bool flag	Render disabled.
`autoContrast`	bool flag	Auto-pick a label color that meets contrast.
`justify`	any CSS `justify-content` value	Distribution of the label + sections row.
`type`	`button` (default), `submit`, `reset`	Button behavior inside a `<form>`.
`id`	string	HTML `id` on the rendered button (or `<a>`).
`onclick`	JS expression string	JavaScript run on click. In Python pass `attr={'onclick': '…'}`.

Sections

Attribute	Valid values	Description
`leftSection`	string	Content rendered to the left of the label.
`rightSection`	string	Content rendered to the right of the label.

Gradient variant (`variant='gradient'`)

Attribute	Valid values	Description
`gradientFrom`	color	Start color.
`gradientTo`	color	End color.
`gradientDeg`	number (degrees)	Angle.

Spacing, color, and sizing (Mantine style system)

Attribute	Valid values	Description
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	size token or CSS value	Margin.
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	size token or CSS value	Padding.
`bg`, `c`	color	Background, text color.
`w`, `h`	size token or CSS value	Width, height.
`miw`, `mih`, `maw`, `mah`	size token or CSS value	Min / max width and height.

CSS Selectors

Each button mounts inside an island wrapper carrying `data-aardvark-island="Button"`; Mantine's Styles API breaks the rendered element into the root, the inner content row, the label, and any icon section.

```
[data-aardvark-island="Button"]  /* the island wrapper */
.mantine-Button-root             /* the <button> / <a> */
.mantine-Button-inner            /* the content row */
.mantine-Button-label            /* the text label */
.mantine-Button-section          /* a left/right icon slot */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered button. The `onclick` shortcut is the common case: set `onclick` to a JavaScript expression and it runs when the button is clicked — including an inline anonymous function called on the spot. In Python, `onclick` rides this same channel — pass `attr={'onclick': '…'}`. The value is any JS that's valid in an `onclick` attribute — a single expression, a comma-chained sequence, or an IIFE for multi-statement logic.

Say hi

Source: Markdown

```
{% button text='Say hi' onclick="(() => alert('Hello world!'))()" %}
```

Source: Python

```
component('aardvark', 'button', text='Say hi',
          attr={'onclick': "(() => alert('Hello world!'))()"})
```

For any other attribute — `data-*`, ARIA, or a full multi-line handler — pass the `attr={…}` dict directly:

Get started

Source: Markdown

```
{% button text='Get started' url='#' attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'button', text='Get started', url='#', attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Stored-XSS surface on multi-author sites

`onclick` ships whatever JavaScript a page author types straight to readers' browsers — it runs unsandboxed and with full access to cookies, storage, and the document. On a single-author site that's fine (you wrote the JS). On a multi-author site, lock it down site-wide with the `attrPolicy` block in `aardvark.config.yaml`: `deny: ['on*']` refuses every inline event handler, or use `allow:`

for an allowlist.

CloseButton

`{% closebutton %}` is a **built-in** tag for a **dismiss button** — the small you put on dialogs, alerts, chips, and panels. It renders a Mantine CloseButton, which ships the glyph and an accessible default name (“Close button”) for free.

Use it as `{% closebutton %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'closebutton', …)`.

Source: Markdown

```
{% closebutton %}
```

Source: Python

```
component('aardvark', 'closebutton')
```

Size and radius

`size` accepts `xs`–`xl`; `radius` accepts `xs`–`xl` or any CSS value.

Source: Markdown

```
{% closebutton size='xs' %}
{% closebutton size='sm' %}
{% closebutton size='md' %}
{% closebutton size='lg' %}
{% closebutton size='xl' radius='xl' %}
```

Source: Python

```
for s in ('xs', 'sm', 'md', 'lg'):
    component('aardvark', 'closebutton', size=s)
component('aardvark', 'closebutton', size='xl', radius='xl')
```

Variant and color

`variant` is `subtle` (default) or `transparent`; `color` takes any theme or CSS color.

Source: Markdown

```
{% closebutton variant='subtle' %}
{% closebutton variant='transparent' %}
{% closebutton color='red' %}
{% closebutton color='grape' %}
```

Source: Python

```
component('aardvark', 'closebutton', variant='subtle')
component('aardvark', 'closebutton', variant='transparent')
component('aardvark', 'closebutton', color='red')
component('aardvark', 'closebutton', color='grape')
```

Custom icon

By default the button is a . Pass an `icon` — any `{% icon %}` spec — to replace it (e.g. a trash can for a delete action). When the glyph changes meaning, set a `label` so screen readers name it correctly. Add `filled` for the filled Tabler style of a bare Tabler name.

Remove

Dismiss

Source: Markdown

```
{% closebutton icon='trash' label='Remove' %}
{% closebutton icon='x' label='Dismiss' %}
```

Source: Python

```
component('aardvark', 'closebutton', icon='trash', label='Remove')
component('aardvark', 'closebutton', icon='x', label='Dismiss')
```

Accessible name

Mantine names the button “Close button” by default. Pass a `label` to override it when the button does something more specific (the example above names it “Remove”).

Disabled

Source: Markdown

```
{% closebutton disabled=true %}
```

Source: Python

```
component('aardvark', 'closebutton', disabled=True)
```

With other components

A close button belongs in the corner of a dismissible surface — here in the top-right of a callout, laid out with a `{% group %}`:

Heads up — you have unsaved changes.

Dismiss notice

Source: Markdown

```
{% callout severity='info' %}
{% group justify='space-between' %}
Heads up — you have unsaved changes.
{% closebutton label='Dismiss notice' %}
{% endGroup %}
{% endCallout %}
```

Source: Python

```
component('aardvark', 'closebutton', label='Dismiss notice')
```

Attributes

Omit any attribute to take its Mantine default. Bare boolean flags (`filled`, `disabled`) set the option to `True`; in Python pass `=True`.

Attribute	Valid values	Description
`icon`	Tabler name / Font Awesome class / image path / emoji / inline `<svg>`	An icon spec to replace the default . Read like `{% icon %}`.
`filled`	bool flag	Use the filled Tabler style for a bare Tabler name.
`label`	string	Accessible name (`aria-label`). Defaults to “Close button”.
`variant`	`subtle` (default), `transparent`	Visual style.
`color`	theme color name or CSS color	Button color.
`size`	`xs`–`xl`	Button size.
`radius`	`xs`–`xl` or any CSS value	Corner radius.

`disabled`	bool flag	Render disabled.
`id`	string	HTML `id` on the rendered button.
`onclick`	JS expression string	JavaScript run on click. In Python pass `attr={'onclick': '…'}`.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	size token or CSS value	Margin (Mantine spacing system).

CSS Selectors

Each close button mounts inside an island wrapper carrying `data-aardvark-island="CloseButton"`; Mantine's Styles API exposes the button root.

```
[data-aardvark-island="CloseButton"]  /* the island wrapper */
.mantine-CloseButton-root             /* the × <button> */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered button — the same channel the `onclick` shortcut rides.

Source: Markdown

```
{% closebutton attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'closebutton', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

CopyButton

`{% copybutton %}` is a **built-in** tag for a **copy-to-clipboard button**. Set `value` to the text to copy; on click it writes that value to the clipboard and briefly swaps its label to a confirmation that the copy succeeded.

Use it as `{% copybutton %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'copybutton', …)`.

Copy command

Source: Markdown

```
{% copybutton value='npm install aardvark' label='Copy command' %}
```

Source: Python

```
component('aardvark', 'copybutton', value='npm install aardvark', label='Copy command')
```

Render-prop component, simplified

Mantine's `CopyButton` is a **render-prop** component — its child is a `({ copied, copy }) => …` function, which a build-time Markdown tag can't author. This tag provides the common case for you: a button (with a hover tooltip) that copies `value` and confirms. For a fully custom trigger, write a snippet — a small React component where you have the render-prop in hand.

Labels and timeout

`label` is the resting text; `copiedLabel` is shown after a copy; `timeout` (ms) controls how long the confirmation stays before reverting.

Copy email

Source: Markdown

```
{% copybutton value='hello@example.com' label='Copy email' copiedLabel='Copied!'
timeout=2000 %}
```

Source: Python

```
component('aardvark', 'copybutton', value='hello@example.com', label='Copy email',
          copiedLabel='Copied!', timeout=2000)
```

Appearance

`variant`, `color`, `size`, and `radius` style the button (it turns teal on a successful copy regardless).

light

outline grape

lg + xl radius

Source: Markdown

```
{% copybutton value='one' variant='light' label='light' %}
{% copybutton value='two' variant='outline' color='grape' label='outline grape' %}
{% copybutton value='three' size='lg' radius='xl' label='lg + xl radius' %}
```

Source: Python

```
component('aardvark', 'copybutton', value='one', variant='light', label='light')
component('aardvark', 'copybutton', value='two', variant='outline', color='grape',
label='outline grape')
component('aardvark', 'copybutton', value='three', size='lg', radius='xl', label='lg + xl
radius')
```

With other components

Drop a copy button next to a value you want readers to grab — here paired with the value in a code span inside a `{% group %}`:

aardvark_live_a1b2c3

Copy key

Source: Markdown

```
{% group gap='xs' %}
{% code %}aardvark_live_a1b2c3{% endCode %}
{% copybutton value='aardvark_live_a1b2c3' label='Copy key' copiedLabel='Copied' size='xs'
%}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'copybutton', value='aardvark_live_a1b2c3',
          label='Copy key', copiedLabel='Copied', size='xs')
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`value`	string	The text copied to the clipboard.

`label`	string	Resting button text.
`copiedLabel`	string	Text shown briefly after a copy.
`timeout`	integer (milliseconds)	How long the confirmation stays before reverting.
`variant`	`filled`, `light`, `outline`, `subtle`, `default`, `transparent`, `white`	Visual style.
`color`	theme color name or CSS color	Button color.
`size`	`xs`–`xl`	Button size.
`radius`	`xs`–`xl` or any CSS value	Corner radius.

For a custom trigger element, write a snippet — inside your own React component you get Mantine's `CopyButton` render-prop in full.

CSS Selectors

The button mounts inside an island wrapper carrying `data-aardvark-island="CopyButton"` and renders a Mantine `Button`, so Mantine's Button Styles API classes apply to the rendered element.

```
[data-aardvark-island="CopyButton"]  /* the island wrapper */
.mantine-Button-root                 /* the rendered <button> */
.mantine-Button-label                /* the resting / confirmation text */
.mantine-Button-inner                /* the content row */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered button.

Copy command

Source: Markdown

```
{% copybutton value='npm install aardvark' label='Copy command' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'copybutton', value='npm install aardvark', label='Copy command',
```

```
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

FileButton

`{% filebutton %}` is a **built-in** tag for a **file-picker button**. Clicking it opens the browser's native file dialog; the chosen file name appears beside the button.

Use it as `{% filebutton %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'filebutton', …)`.

Upload a file

Source: Markdown

```
{% filebutton label='Upload a file' %}
```

Source: Python

```
component('aardvark', 'filebutton', label='Upload a file')
```

Demo only — no upload

This tag is a **client-side demo**: it opens the picker and shows the selected file name, but it does **not** upload anywhere — wiring a file to a server needs JavaScript you write. Like `CopyButton`, Mantine's `FileButton` is a **render-prop** component (its child is a `(props) => …` function), which a build-time Markdown tag can't author, so this tag supplies the picker-and-name UI for you. For real uploads, write a snippet — a small React component where you hold the render-prop and can send the chosen file wherever you like.

Limit file types

`accept` is a standard HTML accept string — MIME types or extensions, comma-separated.

Pick an image

Source: Markdown

```
{% filebutton label='Pick an image' accept='image/png,image/jpeg' %}
```

Source: Python

```
component('aardvark', 'filebutton', label='Pick an image', accept='image/png,image/jpeg')
```

Multiple files

Set `multiple` to let the picker select several files at once.

Pick files

Source: Markdown

```
{% filebutton label='Pick files' multiple=true %}
```

Source: Python

```
component('aardvark', 'filebutton', label='Pick files', multiple=True)
```

Appearance

`variant`, `color`, `size`, and `radius` style the button.

light

outline grape

lg

Source: Markdown

```
{% filebutton label='light' variant='light' %}
{% filebutton label='outline grape' variant='outline' color='grape' %}
{% filebutton label='lg' size='lg' radius='xl' %}
```

Source: Python

```
component('aardvark', 'filebutton', label='light', variant='light')
component('aardvark', 'filebutton', label='outline grape', variant='outline', color='grape')
component('aardvark', 'filebutton', label='lg', size='lg', radius='xl')
```

With other components

Pair the picker with a hint in a card to frame what's expected:

Attach your avatar

PNG or JPEG, up to 2 MB.

Choose image

Source: Markdown

```
{% card title='Attach your avatar' %}
PNG or JPEG, up to 2 MB.

{% filebutton label='Choose image' accept='image/png,image/jpeg' variant='light' %}
```

```
{% endCard %}
```

Source: Python

```
component('aardvark', 'filebutton', label='Choose image',
          accept='image/png,image/jpeg', variant='light')
```

Attributes

Omit any attribute to take its default. The bare boolean flag `multiple` sets the option to `True`; in Python pass `=True`.

Attribute	Valid values	Description
`label`	string	Button text.
`accept`	HTML accept string (MIME types / extensions, comma-separated)	Limit the picker to certain file types.
`multiple`	bool flag	Allow selecting several files.
`variant`	`filled`, `light`, `outline`, `subtle`, `default`, `transparent`, `white`	Visual style.
`color`	theme color name or CSS color	Button color.
`size`	`xs`–`xl`	Button size.
`radius`	`xs`–`xl` or any CSS value	Corner radius.

For a custom trigger or to actually handle the chosen file, write a snippet — inside your own React component you get Mantine's `FileButton` render-prop in full.

CSS Selectors

The picker mounts inside an island wrapper carrying `data-aardvark-island="FileButton"`; it renders a Mantine `Button` next to the chosen file name, laid out in a Mantine `Group`.

```
[data-aardvark-island="FileButton"]  /* the island wrapper */
.mantine-Group-root                  /* the button + filename row */
.mantine-Button-root                 /* the picker <button> */
.mantine-Button-label                /* the button text */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered button.

Upload a file

Source: Markdown

```
{% filebutton label='Upload a file' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'filebutton', label='Upload a file', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

UnstyledButton

`{% unstyledbutton %}` is a **built-in** tag for a **button with no styles** — Mantine's UnstyledButton. It renders a real, accessible `<button>` (keyboard-focusable, correct semantics) with **none** of Mantine's button chrome, so you can paint it however you like with the Mantine style props or your own CSS.

Use it as `{% unstyledbutton %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'unstyledbutton', …)`.

A plain, accessible button

Source: Markdown

```
{% unstyledbutton text='A plain, accessible button' %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', text='A plain, accessible button')
```

Label

Give the label inline with `text`, or as the block body when it wraps other markup. In Python, pass the body as `children`.

Inline label

Block label

Source: Markdown

```
{% unstyledbutton text='Inline label' %}
{% unstyledbutton %}Block label{% endUnstyledbutton %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', text='Inline label')
component('aardvark', 'unstyledbutton', children='Block label')
```

Style it yourself

The whole point is that you supply the look. The Mantine style props are forwarded — padding (`p`, `px`, …), background (`bg`), text color (`c`), sizing (`w`, `h`), and the rest — so you can build a custom button inline:

Custom

Bordered

Source: Markdown

```
{% unstyledbutton text='Custom' bg='grape.6' c='white' px='md' py='xs' %}
{% unstyledbutton text='Bordered' c='grape' px='md' py='xs' %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', text='Custom', bg='grape.6', c='white', px='md',
py='xs')
component('aardvark', 'unstyledbutton', text='Bordered', c='grape', px='md', py='xs')
```

For anything more involved, reach for a snippet or a CSS class via `id`.

Link mode

Set `url` and it renders as a link (`<a href>`) instead of a `<button>`: it navigates on click and works without JavaScript. `target`, `rel`, and `download` pass through.

Go to components

Source: Markdown

```
{% unstyledbutton text='Go to components' url='/components/' c='grape' %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', text='Go to components', url='/components/',
c='grape')
```

With other components

Because it carries no chrome of its own, an unstyled button is the right base for a fully custom clickable surface — here a whole row in a card, styled inline:

Recent activity

View all 24 events →

Source: Markdown

```
{% card title='Recent activity' %}
{% unstyledbutton url='/components/' c='grape' p='xs' w='100%' %}View all 24 events →{%
endUnstyledbutton %}
```

```
{% endCard %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', children='View all 24 events →',
          url='/components/', c='grape', p='xs', w='100%')
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`text`	string	Label, when not using the block body.
`url`	relative path or `http(s)://` URL	Render as a link (`<a href>`). `javascript:`/`data:`/`vbscript:`/`file:`/`blob:` schemes are rejected at build time.
`target`	e.g. `_blank`	Where to open the link (link mode).
`rel`	e.g. `noopener noreferrer`	Link relationship (link mode).
`download`	filename string	Suggest a filename for the linked file (link mode).
`id`	string	HTML `id` on the rendered button (or `<a>`).
`onclick`	JS expression string	JavaScript run on click. In Python pass `attr={'onclick': '…'}`.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	size token or CSS value	Margin (Mantine spacing system).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	size token or CSS value	Padding (Mantine spacing system).
`bg`, `c`	color	Background, text color.
`w`, `h`, `miw`, `mih`, `maw`, `mah`	size token or CSS value	Width / height / min / max.

CSS Selectors

Each button mounts inside an island wrapper carrying `data-aardvark-island="UnstyledButton"`; Mantine's Styles API exposes the (chrome-free) root element.

```
[data-aardvark-island="UnstyledButton"]  /* the island wrapper */
.mantine-UnstyledButton-root             /* the rendered <button> / <a> */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered button. The `onclick` shortcut is the common case: set `onclick` to a JavaScript expression and it runs on click. In Python, `onclick` rides this same channel — pass `attr={'onclick': '…'}`.

Click me

Source: Markdown

```
{% unstyledbutton text='Click me' onclick="(() => alert('Hi!'))()" c='grape' %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', text='Click me', c='grape',
          attr={'onclick': "(() => alert('Hi!'))()"})
```

For any other attribute — `data-*`, ARIA, or a full multi-line handler — pass the `attr={…}` dict directly:

A plain, accessible button

Source: Markdown

```
{% unstyledbutton text='A plain, accessible button' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'unstyledbutton', text='A plain, accessible button', attr={'onclick':
'''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

(See the `{% button %}` page's note about `onclick` as a stored-XSS surface on multi-author sites, and the site-wide `attrPolicy`.)

Navigation

Tags for moving around: links, trails, toggles, and on-page guides. Each ships with Aardvark, so it's a single tag with no setup — and each wraps a Mantine component, so the full option set is there when you need it.

- Anchor — a styled text link with the full Mantine Text surface (underline, color, weight, gradient).
- Breadcrumbs — a breadcrumb trail from a JSON or compact pipe-delimited list of crumbs.
- Burger — a hamburger / cross menu toggle.
- NavLink — a navigation link with an icon, description, active state, and one level of nested sub-links.
- Pagination — an interactive pager whose active page tracks clicks out of the box.
- TableOfContents — an on-page contents list that scrapes the current page's headings and highlights the one in view.

These three render automatically from plain Markdown — no tag to learn — but are documented here too:

- Steps — numbered Markdown lists become a vertical Steps timeline.
- Tabs — switch between Markdown panels with a sliding underline and a crossfade.
- Tree — a nested file/folder explorer with collapsible folders and per-file source in a modal.

Anchor

A **built-in** tag for a styled text link, built on Mantine's `Anchor` — a themed `<a>` that carries the full **Text** styling surface. Set the destination with `url` and style the label with `underline`, `c` (color), `size`, `fw` (weight), `inherit`, and a gradient `variant`. The label is the block body or a `text` param.

Use it as `{% anchor %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'anchor', …)`.

Demonstrations

Basic link

The label is the block body (or a `text` param); `url` is the destination. Add `target`/`rel`/`download` for the standard link attributes.

Read the quickstart

.

Mantine

Source: Markdown

```
{% anchor url='/docs/#quickstart' %}Read the quickstart{% endAnchor %}
{% anchor url='https://mantine.dev' text='Mantine' target='_blank' rel='noopener' %}
```

Source: Python

```
component('aardvark', 'anchor', url='/docs/#quickstart', children='Read the quickstart')
component('aardvark', 'anchor', url='https://mantine.dev', text='Mantine', target='_blank',
rel='noopener')
```

Underline modes

`underline` controls when the underline shows: `always`, `hover` (the default), `never`, or `not-hover`.

always

.

hover

.

never

.

not-hover

Source: Markdown

```
{% anchor url='/' underline='always' %}always{% endAnchor %}
{% anchor url='/' underline='hover' %}hover{% endAnchor %}
{% anchor url='/' underline='never' %}never{% endAnchor %}
{% anchor url='/' underline='not-hover' %}not-hover{% endAnchor %}
```

Source: Python

```
for mode in ('always', 'hover', 'never', 'not-hover'):
    component('aardvark', 'anchor', url='/', underline=mode, children=mode)
```

Color, size, weight, and clamp

Style the text with `c` (any theme color or `dimmed`), `size` (`xs`–`xl`), `fw` (font weight), `inherit` (match the surrounding text's font), and `lineClamp` (clamp to N lines with an ellipsis).

Bold grape

.

Small dimmed

.

Extra large

Source: Markdown

```
{% anchor url='/' c='grape' fw='700' %}Bold grape{% endAnchor %}
{% anchor url='/' c='dimmed' size='xs' %}Small dimmed{% endAnchor %}
{% anchor url='/' size='xl' %}Extra large{% endAnchor %}
```

Source: Python

```
component('aardvark', 'anchor', url='/', c='grape', fw='700', children='Bold grape')
component('aardvark', 'anchor', url='/', c='dimmed', size='xs', children='Small dimmed')
component('aardvark', 'anchor', url='/', size='xl', children='Extra large')
```

Gradient label

Set `variant='gradient'` and supply the gradient endpoints with `gradientFrom`, `gradientTo`, and the angle `gradientDeg`.

Gradient

Source: Markdown

```
{% anchor url='/' variant='gradient' gradientFrom='indigo' gradientTo='cyan' gradientDeg=90
fw='700' %}Gradient{% endAnchor %}
```

Source: Python

```
component(
    'aardvark', 'anchor', url='/', variant='gradient',
    gradientFrom='indigo', gradientTo='cyan', gradientDeg=90, fw='700',
    children='Gradient',
)
```

With other components

Anchors sit naturally inside other components — here, one closes out a `{% callout %}` as the call to action.

Next steps

Ready to dive in?

Open the quickstart

.

Source: Markdown

```
{% callout severity='info' title='Next steps' %}
Ready to dive in? {% anchor url='/docs/#quickstart' fw='700' %}Open the quickstart{%
endAnchor %}.
{% endCallout %}
```

Source: Python

```
link = component('aardvark', 'anchor', url='/docs/#quickstart', fw='700', children='Open the
quickstart')
component('aardvark', 'callout', severity='info', title='Next steps', children=f'Ready to
dive in? {link}.')
```

Attributes

Omit any attribute to take its default. An `url` using a `javascript:`, `data:`, `vbscript:`, `file:`, or `blob:` scheme is rejected at build time — use a relative path or an `http(s)://` link.

Attribute	Valid values	Description

`url`	A relative path or `http(s)://` URL (**required**)	The link destination (the `href`). Unsafe schemes are build-rejected.
`text`	Any string	The label, when not using the block body.
`target`	e.g. `_blank`, `_self`	Standard link `target` attribute.
`rel`	e.g. `noopener`, `nofollow`	Standard link `rel` attribute.
`download`	Filename, or a bare flag	Standard link `download` attribute.
`underline`	`always`, `hover` (default), `never`, `not-hover`	When the underline appears.
`c`	Any theme color (`blue`, `grape`, ...) or `dimmed`	Text color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS size	Font size.
`fw`	A font weight (e.g. `500`, `700`)	Font weight.
`inherit`	`true` / `false` (default)	Inherit the surrounding text's font styles instead of Mantine's.
`lineClamp`	An integer	Clamp the label to N lines with an ellipsis.
`variant`	`gradient`	Render the label as a gradient (pair with the gradient props below).
`gradientFrom`	Any theme color	Gradient start color (with `variant='gradient'`).
`gradientTo`	Any theme color	Gradient end color (with `variant='gradient'`).
`gradientDeg`	A number (degrees)	Gradient angle (with `variant='gradient'`).

CSS Selectors

Each anchor mounts inside an island wrapper carrying `data-aardvark-island="Anchor"`, and the link itself is a Mantine `Anchor` — target either the wrapper or Mantine's Styles API class to restyle every anchor on the page.

```
[data-aardvark-island="Anchor"]  /* the island wrapper */
```

```
.mantine-Anchor-root              /* the rendered <a> */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered link.

Read the quickstart

Source: Markdown

```
{% anchor url='/docs/#quickstart' attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Read the quickstart{% endAnchor %}
```

Source: Python

```
component('aardvark', 'anchor', url='/docs/#quickstart', attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}, children='Read the quickstart')
```

Breadcrumbs

A **built-in** tag for a breadcrumb trail, built on Mantine's `Breadcrumbs`. Pass the crumbs in `items` — either a compact comma-separated list of `Label | /href` pairs or a JSON array of `{label, href}` objects — and set the `separator`. A crumb with no `href` (typically the last, current one) renders as plain "you are here" text.

Use it as `{% breadcrumbs %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'breadcrumbs', …)`.

Demonstrations

Compact form

The **compact** form is a comma-separated list of `Label | /href` pairs. Drop the `| href` on a crumb to render it as plain text — the last (current) crumb usually has no link.

Source: Markdown

```
{% breadcrumbs items='Home | /, Components | /components/, Breadcrumbs' %}
```

Source: Python

```
component('aardvark', 'breadcrumbs', items='Home | /, Components | /components/,
Breadcrumbs')
```

JSON form

The **JSON** form is an array of `{label, href}` objects — handy when a label itself contains a comma or a pipe. Omit `href` to render a crumb as plain text.

Source: Markdown

```
{% breadcrumbs
items='[{"label":"Home","href":"/"},{"label":"Docs","href":"/getting-started/"},{"label":"Th
is page"}]' %}
```

Source: Python

```
component(
    'aardvark', 'breadcrumbs',
    items='[{"label":"Home","href":"/"},{"label":"Docs","href":"/getting-started/"},{"label":"Th
is page"}]',
)
```

Separator and spacing

`separator` sets the character drawn between crumbs (defaults to /); `separatorMargin` sets the space around it — a Mantine size (`xs`–`xl`) or any CSS value.

Source: Markdown

```
{% breadcrumbs items='Home | /, Guides | /getting-started/, Setup' separator='→'
separatorMargin='md' %}
```

Source: Python

```
component(
    'aardvark', 'breadcrumbs',
    items='Home | /, Guides | /getting-started/, Setup',
    separator='→', separatorMargin='md',
)
```

With other components

Build a trail in Python from a list of pages — the loop-friendly form — and pair it with other built-ins. Here the crumbs are assembled into the compact string before the tag is called.

Source: Markdown

```
{% breadcrumbs items='Home | /, Components | /components/, Navigation |
/components/navigation/, Breadcrumbs' separator='/' %}
```

Source: Python

```
trail = [
    ('Home', '/'),
    ('Components', '/components/'),
    ('Navigation', '/components/navigation/'),
    ('Breadcrumbs', None),  # current page — no href
]
items = ', '.join(f'{label} | {href}' if href else label for label, href in trail)
component('aardvark', 'breadcrumbs', items=items, separator='/')
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description

`items`	JSON array of `{label, href}`, **or** a comma-separated list of `Label | /href` pairs (**required**)	The crumbs. A crumb with no `href` (or the last crumb) renders as plain, dimmed “you are here” text.
`separator`	Any string (defaults to `/`)	The character drawn between crumbs.
`separatorMargin`	A Mantine size (`xs`–`xl`) or any CSS value	The space around the separator.

CSS Selectors

The trail renders inside an island wrapper carrying `data-aardvark-island="Breadcrumbs"`, with Mantine’s Styles API classes on the row, each crumb, and the separator between them.

```
[data-aardvark-island="Breadcrumbs"]  /* the island wrapper */
.mantine-Breadcrumbs-root             /* the crumb row */
.mantine-Breadcrumbs-breadcrumb       /* a single crumb */
.mantine-Breadcrumbs-separator        /* the divider between crumbs */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered trail.

Source: Markdown

```
{% breadcrumbs items='Home | /, Components | /components/, Breadcrumbs' attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'breadcrumbs', items='Home | /, Components | /components/,
Breadcrumbs', attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Burger

A **built-in** tag for the classic hamburger menu toggle, built on Mantine's `Burger`. Set `opened=true` to render the **cross** (an open state) instead of the three lines, and tune `size`, `color`, `lineSize` (bar thickness), and the morph transition. It is presentational on its own — wire interactivity with an `onclick` handler or drive it from a snippet.

Use it as `{% burger %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'burger', …)`.

Demonstrations

Closed and open

Without `opened`, the burger shows the three lines. Set `opened=true` to show the cross. Always give it a `label` — it becomes the `aria-label`, since the button has no visible text.

Source: Markdown

```
{% burger label='Open navigation' %}
{% burger opened=true label='Close navigation' %}
```

Source: Python

```
component('aardvark', 'burger', label='Open navigation')
component('aardvark', 'burger', opened=True, label='Close navigation')
```

Sizes, colors, and line thickness

`size` takes `xs`–`xl` or any CSS size; `color` takes any theme color; `lineSize` sets the bar thickness in pixels.

Source: Markdown

```
{% burger size='sm' label='Menu' %}
{% burger size='lg' color='grape' label='Menu' %}
{% burger size='xl' color='teal' lineSize=4 label='Menu' %}
```

Source: Python

```
component('aardvark', 'burger', size='sm', label='Menu')
component('aardvark', 'burger', size='lg', color='grape', label='Menu')
component('aardvark', 'burger', size='xl', color='teal', lineSize=4, label='Menu')
```

Transition tuning

`transitionDuration` (ms) and `transitionTimingFunction` (any CSS timing function) control the burger↔cross morph.

Source: Markdown

```
{% burger opened=true color='indigo' transitionDuration=600
transitionTimingFunction='ease-in-out' label='Close menu' %}
```

Source: Python

```
component(
    'aardvark', 'burger', opened=True, color='indigo',
    transitionDuration=600, transitionTimingFunction='ease-in-out',
    label='Close menu',
)
```

With other components

Burger is presentational on its own. To wire a real click, forward an `onclick` through the island `attr={...}` channel; for true open/close state, drive it from a snippet where you own the React state and pass `opened`/`onClick` to `component('Burger', ...)`. Here it sits in a `{% group %}` next to a label.

Navigation

Source: Markdown

```
{% group %}
{% burger label='Toggle navigation' attr={'onclick':
'document.body.classList.toggle(\'nav-open\')'} %}
{% anchor url='/components/navigation/' c='dimmed' %}Navigation{% endAnchor %}
{% endGroup %}
```

Source: Python

```
burger = component('aardvark', 'burger', label='Toggle navigation', attr={'onclick':
"document.body.classList.toggle('nav-open')"})
link = component('aardvark', 'anchor', url='/components/navigation/', c='dimmed',
children='Navigation')
component('aardvark', 'group', children=burger + link)
```

Attributes

Omit any attribute to take its default. Burger renders a `<button>` and takes no body.

Attribute	Valid values	Description
`opened`	`true` / `false` (default)	`true` shows the cross (open state); otherwise the three-line burger (closed).
`size`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS size	Overall size of the control.
`color`	Any theme color (`blue`, `grape`, ...)	Color of the bars.
`lineSize`	An integer (pixels)	Thickness of each bar.
`transitionDuration`	An integer (milliseconds)	Duration of the burger↔cross morph.
`transitionTimingFunction`	A CSS timing function (`ease`, `ease-in-out`, ...)	Easing of the morph.
`label`	Any string	Accessible name — rendered as `aria-label` (the button has no visible text).

CSS Selectors

The control mounts inside an island wrapper carrying `data-aardvark-island="Burger"`; Mantine's Styles API exposes the button root and the morphing bars.

```
[data-aardvark-island="Burger"]  /* the island wrapper */
.mantine-Burger-root             /* the <button> */
.mantine-Burger-burger           /* the three-line / cross glyph */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered button — the same channel the `onclick` shortcut above rides.

Source: Markdown

```
{% burger label='Open navigation' attr={'onclick': '''
const value = this.getAttribute('aria-label');
console.log('attr demo value:', value);
alert(value);
```

```
'''} %}
```

Source: Python

```
component('aardvark', 'burger', label='Open navigation', attr={'onclick': '''
const value = this.getAttribute('aria-label');
console.log('attr demo value:', value);
alert(value);
'''})
```

NavLink

A **built-in** tag for a navigation link, built on Mantine's `NavLink`: a `label` with an optional `description`, a left/right **icon** (any `{% icon %}` spec), an `active` state, a `variant` and `color`, and one level of **nested** sub-links via `items`.

Use it as `{% navlink %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'navlink', …)`.

Demonstrations

Label, description, icon, and active

The `label` is required; add a `description` for secondary text and `leftSection` / `rightSection` for an icon. Set `active=true` to highlight the current link, and `href` to render it as a real `<a>`.

Dashboard

Active page

Source: Markdown

```
{% navlink label='Dashboard' description='Your overview' leftSection='dashboard'
href='/dashboard/' %}
{% navlink label='Active page' leftSection='star' active=true %}
```

Source: Python

```
component('aardvark', 'navlink', label='Dashboard', description='Your overview',
leftSection='dashboard', href='/dashboard/')
component('aardvark', 'navlink', label='Active page', leftSection='star', active=True)
```

Variants and color

`variant` is `light` (the default), `filled`, or `subtle`; `color` sets the accent for the active/hover state.

Light

Filled

Subtle

Source: Markdown

```
{% navlink label='Light' variant='light' active=true %}
{% navlink label='Filled' variant='filled' active=true color='grape' %}
{% navlink label='Subtle' variant='subtle' active=true %}
```

Source: Python

```
component('aardvark', 'navlink', label='Light', variant='light', active=True)
component('aardvark', 'navlink', label='Filled', variant='filled', active=True,
color='grape')
component('aardvark', 'navlink', label='Subtle', variant='subtle', active=True)
```

Disabled, no-wrap, and right section

`disabled` dims and disables the link; `noWrap` keeps the label on one line; a `rightSection` icon sits at the trailing edge (handy for a chevron or external-link mark).

Disabled

External docs

Source: Markdown

```
{% navlink label='Disabled' leftSection='lock' disabled=true %}
{% navlink label='External docs' rightSection='external-link' href='https://mantine.dev' %}
```

Source: Python

```
component('aardvark', 'navlink', label='Disabled', leftSection='lock', disabled=True)
component('aardvark', 'navlink', label='External docs', rightSection='external-link',
href='https://mantine.dev')
```

Nested sub-links

Pass nested sub-links as a JSON array in `items` — `children` is reserved for the block body, so the nested data rides its own param. Set `defaultOpened=true` to start the group expanded and `childrenOffset` to indent it. Nesting goes **one level** deep through the tag.

Settings

Source: Markdown

```
{% navlink label='Settings' leftSection='settings' childrenOffset=20 defaultOpened=true
items='[{"label":"Profile","href":"/profile/","leftSection":"user"},{"label":"Billing","href
":"/billing/","leftSection":"credit-card"}]' %}
```

Source: Python

```
import json
items = json.dumps([
    {'label': 'Profile', 'href': '/profile/', 'leftSection': 'user'},
    {'label': 'Billing', 'href': '/billing/', 'leftSection': 'credit-card'},
])
component('aardvark', 'navlink', label='Settings', leftSection='settings',
childrenOffset=20, defaultOpened=True, items=items)
```

With other components

Build a navigation column by looping over a list of pages in Python and emitting a `{% navlink %}` for each, marking the current page `active`. Wrap the run in a `{% stack %}` for vertical spacing.

Overview

Navigation

Layout

Source: Markdown

```
{% stack gap='xs' %}
{% navlink label='Overview' leftSection='home' href='/' %}
{% navlink label='Navigation' leftSection='compass' active=true %}
{% navlink label='Layout' leftSection='layout' href='/components/' %}
{% endStack %}
```

Source: Python

```
pages = [
    ('Overview', 'home', '/'),
    ('Navigation', 'compass', None),  # current page — active
    ('Layout', 'layout', '/components/'),
]
links = ''
for label, icon, href in pages:
    links += component('aardvark', 'navlink', label=label, leftSection=icon,
                       href=href or '', active=href is None)
component('aardvark', 'stack', gap='xs', children=links)
```

For deeper trees or fully custom link state, build a snippet that nests `component('NavLink', ...)` directly.

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`label`	Any string (**required**)	The link text.
`description`	Any string	Secondary text below the label.
`href`	A relative path or URL	Destination — renders the link as an `<a>`.

`leftSection`	An icon spec (Tabler name, Font Awesome class, image path, emoji, or inline SVG)	Leading icon, exactly like `{% icon %}`.
`rightSection`	An icon spec (as above)	Trailing icon.
`active`	`true` / `false` (default)	Highlight this link as the current one.
`variant`	`light` (default), `filled`, `subtle`	Visual style of the active/hover state.
`color`	Any theme color (`blue`, `grape`, ...)	Accent color for the active/hover state.
`disabled`	`true` / `false` (default)	Dim and disable interaction.
`noWrap`	`true` / `false` (default)	Keep the label on one line.
`autoContrast`	`true` / `false` (default)	Auto-pick a readable label color against `color`.
`items`	JSON array of nested-link objects (`{label, href, leftSection, …}`)	One level of nested sub-links.
`childrenOffset`	An integer (pixels)	Indent of the nested group.
`defaultOpened`	`true` / `false` (default)	Whether the nested group starts expanded.

CSS Selectors

Each link mounts inside an island wrapper carrying `data-aardvark-island="NavLink"`; Mantine's Styles API breaks the rendered element into the root, label, description, and (for nested links) the chevron.

```
[data-aardvark-island="NavLink"]  /* the island wrapper */
.mantine-NavLink-root             /* the link/button */
.mantine-NavLink-label            /* the primary label */
.mantine-NavLink-description      /* the secondary line */
.mantine-NavLink-chevron          /* the expand chevron (nested links) */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered link.

Dashboard

Source: Markdown

```
{% navlink label='Dashboard' description='Your overview' leftSection='dashboard'
href='/dashboard/' attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'navlink', label='Dashboard', description='Your overview',
leftSection='dashboard', href='/dashboard/', attr={'onclick': '''
event.preventDefault();
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Pagination

A **built-in** tag for an interactive pager, built on Mantine's `Pagination`. Set `total` (the number of pages) and the active page **tracks clicks out of the box** — the tag ships the client state Mantine's controlled component needs. Tune the shape with `siblings`, `boundaries`, `withEdges`, and `withControls`, and the look with `size`, `radius`, `color`, and `gap`.

Use it as `{% pagination %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'pagination', …)`.

Demonstrations

Basic pager

Only `total` is required. Seed the initial page with `defaultValue` (defaults to `1`). Click a page below — the active state follows.

Source: Markdown

```
{% pagination total=10 %}
```

Source: Python

```
component('aardvark', 'pagination', total=10)
```

Shape — siblings, boundaries, and edges

`siblings` sets how many page buttons flank the current page; `boundaries` sets how many are pinned at each end; `withEdges` adds first/last jump buttons. `defaultValue` picks the page shown on load.

Source: Markdown

```
{% pagination total=20 defaultValue=10 siblings=2 boundaries=2 withEdges=true color='grape'
radius='xl' %}
```

Source: Python

```
component(
    'aardvark', 'pagination', total=20, defaultValue=10,
    siblings=2, boundaries=2, withEdges=True, color='grape', radius='xl',
)
```

Controls off

`withControls` is on by default (the prev/next arrows). Turn it off with `withControls=false`; `disabled` dims and disables the whole control.

Source: Markdown

```
{% pagination total=8 withControls=false %}
{% pagination total=8 disabled=true %}
```

Source: Python

```
component('aardvark', 'pagination', total=8, withControls=False)
component('aardvark', 'pagination', total=8, disabled=True)
```

Sizes, radius, and gap

`size` and `radius` take `xs`–`xl`; `gap` sets the space between buttons (a Mantine size or any CSS value).

Source: Markdown

```
{% pagination total=5 size='sm' %}
{% pagination total=5 size='lg' gap='xl' %}
```

Source: Python

```
component('aardvark', 'pagination', total=5, size='sm')
component('aardvark', 'pagination', total=5, size='lg', gap='xl')
```

With other components

Compute the page count in Python from a collection and a page size, then render the pager. Here the pager sits under a `{% stack %}` caption.

Page 1 of 4

Source: Markdown

```
{% stack gap='xs' align='center' %}
Page 1 of 4
{% pagination total=4 color='indigo' %}
{% endStack %}
```

Source: Python

```
import math
```

```
items, per_page = 37, 10
total = math.ceil(items / per_page)  # -> 4 pages
caption = f'Page 1 of {total}'
pager = component('aardvark', 'pagination', total=total, color='indigo')
component('aardvark', 'stack', gap='xs', align='center', children=caption + pager)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`total`	A positive integer (**required**)	Number of pages.
`defaultValue`	An integer (defaults to `1`)	The page shown on load.
`siblings`	An integer	Page buttons on each side of the current page.
`boundaries`	An integer	Page buttons pinned at the start and end.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Size of the buttons.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius of the buttons.
`color`	Any theme color (`blue`, `grape`, …)	Active-page color.
`gap`	A Mantine size or any CSS value	Space between buttons.
`withEdges`	`true` / `false` (default)	Show first/last jump buttons.
`withControls`	`true` (default) / `false`	Show the prev/next arrows.
`disabled`	`true` / `false` (default)	Dim and disable the whole control.
`autoContrast`	`true` / `false` (default)	Auto-pick a readable label color against `color`.

CSS Selectors

The control mounts inside an island wrapper carrying `data-aardvark-island="Pagination"`; Mantine's Styles API exposes the row, each page button, and the truncation dots.

```
[data-aardvark-island="Pagination"]  /* the island wrapper */
.mantine-Pagination-root             /* the button row */
.mantine-Pagination-control          /* a single page / arrow button */
.mantine-Pagination-dots             /* the … truncation marker */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered control.

Source: Markdown

```
{% pagination total=10 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'pagination', total=10, attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Steps

Write an ordinary numbered list in Markdown and Aardvark renders it as a Steps block: a vertical run of numbered badges joined by a connecting line, each badge beside that step's content, built from Mantine's `Timeline`. Each step's body is full Markdown — paragraphs, code, images, even nested lists.

Steps is not a tag. There is nothing to write as `{% steps %}` and no `component('aardvark', 'steps', …)` call — a plain `1. … 2. … 3.` numbered list is the entire authoring surface, and Aardvark transforms it at build time. The behavior is controlled site-wide by `steps: false` in `aardvark.config.yaml`.

Default

Just write a numbered list. Lead each step with a bold phrase to give it a title, or write plain prose — both read well beside the number.

Preview

1. **Install** — pull in the toolchain with `npm install`.
2. **Configure** — add an `aardvark.config.yaml` and point `content` at your Markdown.
3. **Build** — run `vark build`, then serve the `build/` directory.

Source: Markdown

```
1. **Install** — pull in the toolchain with `npm install`.
2. **Configure** — add an `aardvark.config.yaml` and point `content` at your Markdown.
3. **Build** — run `vark build`, then serve the `build/` directory.
```

Multi-block steps

A step can hold any block content — extra paragraphs, a code block, an image. Leave a blank line between items (a "loose" list) and give each step as much room as it needs.

Preview

1. Create the config.

```
site:
  name: My Docs
```

2. Add your first page at `content/index.md`.
3. Build and serve.

Source: Markdown

````
1. Create the config.

   ```yaml
   site:
     name: My Docs
   ```

2. Add your first page at `content/index.md`.

3. Build and serve.
````

Numbering

The list's own numbering is kept, so a list that starts at `3` shows `3, 4, 5`.

Preview

3. Third
4. Fourth
5. Fifth

Source: Markdown

```
3. Third
4. Fourth
5. Fifth
```

Nested numbered lists

Careful with nested numbered lists

Only a **top-level** numbered list becomes Steps. A numbered list **nested inside a step** isn't rendered as a second timeline — it stays a plain ordered list, which can look awkward tucked inside the steps. If a page genuinely needs numbered **sub-steps**, turn Steps off (`steps: false` in `aardvark.config.yaml`, below) and author the whole list as ordinary Markdown.

With Steps on, a nested numbered list flattens into the step body rather than forming its own timeline. Bullet lists are never Steps, at any level.

Preview

1. A top-level step — rendered as a Step.
1. A nested numbered list stays a plain ordered list…
2. …sitting inside the step body, which can look out of place.
2. Back at the top level.

Source: Markdown

```
1. A top-level step — rendered as a Step.
   1. A nested numbered list stays a plain ordered list…
   2. …sitting inside the step body, which can look out of place.
2. Back at the top level.
```

With other components

Each step's body is full Markdown, so it can host any other tag — for example a `{% callout %}` inside one step. Keep the blank lines between items so the step bodies render as block Markdown.

Preview

1. **Set up** the project directory.

Edit the config.

The config lives at `aardvark.config.yaml` in the project root.

3. **Build** with `vark build`.

Source: Markdown

```
1. **Set up** the project directory.

2. **Edit** the config.

   {% callout severity="info" %}
   The config lives at `aardvark.config.yaml` in the project root.
   {% endCallout %}

3. **Build** with `vark build`.
```

Turning it off

Auto-Steps is on by default. To render numbered lists as plain ordered lists site-wide, set `steps: false` in `aardvark.config.yaml`:

```
# Render numbered lists as plain <ol> instead of Steps.
steps: false
```

Attributes

Steps has no tag and therefore no per-instance attributes. Its only setting is the site-wide config flag.

Setting	Valid values	Description
`steps` (in `aardvark.config.yaml`)	bool (default `true`)	`false` renders numbered lists as plain `<ol>` instead of a Steps timeline, site-wide.

CSS Selectors

A top-level numbered list becomes a client-mounted `Stepper` island (a Mantine `Timeline`); the Timeline root carries `data-aardvark-steps`, so you can target the island wrapper, that root, or Mantine's Timeline Styles API classes.

```
[data-aardvark-island="Stepper"] /* the island wrapper */
[data-aardvark-steps]            /* the Timeline root */
.mantine-Timeline-root           /* the timeline */
.mantine-Timeline-item           /* a single step */
.mantine-Timeline-itemBullet     /* the numbered badge */
.mantine-Timeline-itemBody       /* the step's content */
```

Injecting Attributes

Steps has no tag — a plain numbered list is the whole authoring surface — so there's no `attr={…}` channel and no per-instance attributes. Style it through the CSS classes above, and toggle the whole feature with `steps:` in `aardvark.config.yaml`.

TableOfContents

A built-in tag for an on-page table of contents, built on Mantine's `TableOfContents`. It scrapes the current page's headings at runtime and highlights the one in view as you scroll (scroll-spy), so it always reflects wherever you drop it — there's no list to maintain. The headings come from the live DOM, so the build-time prerender bakes an empty list and the client fills it in on load.

Use it as `{% tableofcontents %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'tableofcontents', …)`.

Default

With no attributes it tracks every `h1`–`h3` inside the page's content and indents each level. The entries below are this page's own headings, and the active one updates as you scroll.

Preview

Source: Markdown

```
{% tableofcontents %}
```

Source: Python

```
component('aardvark', 'tableofcontents')
```

Depth and indentation

`depth` (1–6) caps which heading levels appear; `depthOffset` sets the pixels of indent added per level. Here we include `h1`–`h2` only, with a wider indent.

Preview

Source: Markdown

```
{% tableofcontents depth=2 depthOffset=24 %}
```

Source: Python

```
component('aardvark', 'tableofcontents', depth=2, depthOffset=24)
```

Variant, color, size, and radius

`variant` switches the active-entry treatment (`light`, `filled`, `none`); `color` sets its accent; `size` and `radius` (`xs`–`xl`) scale the labels and the rounding. `autoContrast` picks a readable label color against `color`.

Preview

Source: Markdown

```
{% tableofcontents variant='filled' color='grape' size='lg' radius='md' autoContrast=true %}
```

Source: Python

```
component(
    'aardvark', 'tableofcontents',
    variant='filled', color='grape', size='lg', radius='md', autoContrast=True,
)
```

Custom selector

By default the contents tracks every `h1`–`h{depth}` inside the page's content (`.aardvark-content`), so site chrome (nav and footer) is never picked up. Pass an explicit `selector` to track a different heading set — say only the `h2`s:

Preview

Source: Markdown

```
{% tableofcontents depth=4 variant='filled' color='blue' selector='.aardvark-content h2' %}
```

Source: Python

```
component(
    'aardvark', 'tableofcontents',
    depth=4, variant='filled', color='blue', selector='.aardvark-content h2',
)
```

With other components

Because a table of contents is just another tag, you can drop it inside any block — for example pinned into an `{% accordion %}` section, or beside prose in a `{% card %}`. Here it sits in its own accordion section so it can be folded away.

Preview

On this page

Source: Markdown

```
{% accordion %}
{% accordionSection title="On this page" %}
{% tableofcontents depth=2 %}
```

```
{% endAccordionSection %}
{% endAccordion %}
```

Source: Python

```
toc = component('aardvark', 'tableofcontents', depth=2)
section = component(
    'aardvark', 'accordionSection', title='On this page', children=toc,
)
component('aardvark', 'accordion', children=section)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`depth`	`1`–`6` (int; default `3`)	Highest heading level to include — `3` tracks `h1`–`h3`. Also shapes the default selector.
`depthOffset`	int (pixels)	Indent added per heading level. Omitted → Mantine's default.
`variant`	`light` (default), `filled`, `none`	Treatment of the active entry.
`color`	any Mantine color (e.g. `blue`, `grape`, `#7048e8`)	Accent color for the active entry.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Label size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner radius of the active highlight.
`autoContrast`	bool (default `false`)	Auto-pick a readable label color against `color`.
`selector`	a CSS selector	Headings to track. Overrides the default (`h1–h{depth}` inside `.aardvark-content`).

CSS Selectors

The widget mounts inside an island wrapper carrying `data-aardvark-island="TableOfContents"`; Mantine's Styles API exposes the list root and each heading link.

```
[data-aardvark-island="TableOfContents"]  /* the island wrapper */
.mantine-TableOfContents-root             /* the list of headings */
.mantine-TableOfContents-control          /* a single heading link */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) onto the rendered widget.

Source: Markdown

```
{% tableofcontents attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'tableofcontents', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Tabs

A built-in tag built from Mantine's `Tabs` pieces (`Tabs.List`, `Tabs.Tab`, `Tabs.Panel`) behind one tag pair, with two motions layered on: the active underline slides to the tab you click instead of jumping, and the panel content crossfades in. Each tab's body is ordinary Markdown — headings, lists, code, links, even nested `component(...)` calls — and renders as such. Motion is suppressed for visitors who ask for `prefers-reduced-motion`.

Use it as `{% tabs %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'tabs', …)`. It is a compound block: the `{% tabs %}` wrapper holds one `{% tab label="…" %}` per panel.

Tabs or CodeGroup?

Use **Tabs** for panels of arbitrary Markdown you label yourself — prose, lists, nested components, or code — that stay independent of each other. If your tabs are the same code shown in several languages or files, reach for **CodeGroup** instead: it labels tabs by language automatically, adds a copy / download button per tab, highlights at build time, and **syncs the chosen language across every code group on the page** (and remembers it across page loads). Both are the same tab widget underneath, so they share the `variant` options, the sliding underline, and the crossfade.

Default

Wrap the set in `{% tabs %} … {% endTabs %}` and give each panel its own `{% tab label="…" %} … {% endTab %}`. Pick the tab open on load with `defaultValue` (a tab's `value`, which defaults to a slug of its label). The panels below are deliberately different heights so the crossfade is visible.

Preview

Install

Run `npm install` to pull in the toolchain.

Configure

Add an `aardvark.config.yaml`:

- set the site `title`
- point `content` at your Markdown
- pick a `theme` accent color

Build

Run `vark build`, then serve the `build/` directory.

Source: Markdown

```
{% tabs defaultValue="install" %}
{% tab label="Install" %}
```

```
Run `npm install` to pull in the toolchain.
{% endTab %}
{% tab label="Configure" %}
Add an `aardvark.config.yaml`:

- set the site `title`
- point `content` at your Markdown
- pick a `theme` accent color
{% endTab %}
{% tab label="Build" %}
Run `vark build`, then serve the `build/` directory.
{% endTab %}
{% endTabs %}
```

Source: Python

```
tabs = (
    component('aardvark', 'tab', label='Install',
              children='Run `npm install` to pull in the toolchain.')
    + component('aardvark', 'tab', label='Configure',
                children='Add an `aardvark.config.yaml`.')
    + component('aardvark', 'tab', label='Build',
                children='Run `vark build`, then serve the `build/` directory.')
)
component('aardvark', 'tabs', defaultValue='install', children=tabs)
```

Pills

`variant="pills"` swaps the underline for Mantine's filled pill. The sliding bar turns off automatically — pills already mark the active tab — while the content still crossfades.

Preview

CLI

Build from the terminal with `vark build`.

Library

Import `aardvark` and call the builder yourself.

Source: Markdown

```
{% tabs variant="pills" defaultValue="cli" %}
{% tab label="CLI" %}Build from the terminal with `vark build`.{% endTab %}
{% tab label="Library" %}Import `aardvark` and call the builder yourself.{% endTab %}
{% endTabs %}
```

Source: Python

```
tabs = (
```

```
    component('aardvark', 'tab', label='CLI',
              children='Build from the terminal with `vark build`.')
    + component('aardvark', 'tab', label='Library',
                children='Import `aardvark` and call the builder yourself.')
)
component('aardvark', 'tabs', variant='pills', defaultValue='cli', children=tabs)
```

Vertical, colored, and disabled tabs

`orientation="vertical"` stacks the tabs down the side; `color` accents the active tab; `grow` makes the tabs share the row width; and a `disabled` flag on a tab makes it non-interactive.

Preview

Overview

A vertical tab list runs down the left, with panels to its right.

Details

The active tab is tinted with the `color` you set — here, grape.

Coming soon

This panel is unreachable while the tab is disabled.

Source: Markdown

```
{% tabs orientation="vertical" color="grape" defaultValue="overview" %}
{% tab label="Overview" %}
A vertical tab list runs down the left, with panels to its right.
{% endTab %}
{% tab label="Details" %}
The active tab is tinted with the `color` you set — here, grape.
{% endTab %}
{% tab label="Coming soon" disabled %}
This panel is unreachable while the tab is disabled.
{% endTab %}
{% endTabs %}
```

Source: Python

```
tabs = (
    component('aardvark', 'tab', label='Overview',
              children='A vertical tab list runs down the left.')
    + component('aardvark', 'tab', label='Details',
                children='The active tab is tinted with the `color` you set.')
    + component('aardvark', 'tab', label='Coming soon', disabled=True,
                children='This panel is unreachable while the tab is disabled.')
)
component(
```

```
    'aardvark', 'tabs',
    orientation='vertical', color='grape', defaultValue='overview', children=tabs,
)
```

With other components

A tab's body is full Markdown, so it can host any other tag — a `{% tree %}` in one tab, a `{% callout %}` in another. This keeps long, alternative explanations from stacking down the page.

Preview

Files

Example project

src

app.py

README.md

Notes

Each tab can hold a different kind of block.

Source: Markdown

```
{% tabs defaultValue="files" %}
{% tab label="Files" %}
{% tree label="Example project" %}
{% folder name="src" defaultOpen %}
{% file name="app.py" %}{% endFile %}
{% endFolder %}
{% file name="README.md" %}{% endFile %}
{% endTree %}
{% endTab %}
{% tab label="Notes" %}
{% callout severity="info" %}
Each tab can hold a different kind of block.
{% endCallout %}
{% endTab %}
{% endTabs %}
```

Source: Python

```
tree = component(
    'aardvark', 'tree', label='Example project',
    children=component(
        'aardvark', 'folder', name='src', defaultOpen=True,
        children=component('aardvark', 'file', name='app.py'),
    ) + component('aardvark', 'file', name='README.md'),
)
```

```
note = component('aardvark', 'callout', severity='info',
                 children='Each tab can hold a different kind of block.')
tabs = (
    component('aardvark', 'tab', label='Files', children=tree)
    + component('aardvark', 'tab', label='Notes', children=note)
)
component('aardvark', 'tabs', defaultValue='files', children=tabs)
```

Attributes

Attributes on `{% tabs %}` pass straight through as Mantine `Tabs` props, so the full Tabs surface is available.

`{% tabs %}`

Attribute	Valid values	Description
`defaultValue`	a tab's `value`	Which tab is open on load. Use this, not `value` — a bare `value` makes Mantine controlled, so the tabs won't switch on click.
`variant`	`default`, `outline`, `pills`	`default` / `outline` keep the sliding underline; `pills` uses Mantine's filled pill (sliding bar off).
`orientation`	`horizontal` (default), `vertical`	Stack the tabs down the side when `vertical`.
`color`	any Mantine color (e.g. `blue`, `grape`)	Accent color for the active tab.
`radius`	`xs`–`xl`	Corner radius of the tab controls.
`grow`	bool flag	Make the tabs share the full row width.
`justify`	`left`, `center`, `right`, `apart`, …	Alignment of the tab list.
`keepMounted`	bool flag	Keep inactive panels mounted in the DOM.

`{% tab %}`

Attribute	Valid values	Description
`label`	string	The clickable tab button text.
`value`	string	Stable id for the tab (defaults to a slug of the label).
`color`	any Mantine color	Accent color for this individual tab.
`disabled`	bool flag	Render the tab disabled (non-interactive).

CSS Selectors

The set mounts inside an island wrapper carrying `data-aardvark-island="Tabs"`; Mantine's Styles API breaks it into the root, the tab strip, each tab button (and its label), and the panel that shows the active tab's body.

```
[data-aardvark-island="Tabs"]  /* the island wrapper */
.mantine-Tabs-root             /* the whole component */
.mantine-Tabs-list             /* the row of tab buttons */
.mantine-Tabs-tab              /* a single tab button */
.mantine-Tabs-tabLabel         /* a tab's text */
.mantine-Tabs-panel            /* the active tab's body */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered tabs root. (Style it through the CSS parts above; set per-tab options like `color`, `disabled`, or `value` on each `{% tab %}`.)

Install

Run `npm install` to pull in the toolchain.

Configure

Add an `aardvark.config.yaml`.

Source: Markdown

```
{% tabs defaultValue="install" attr={'data-analytics': 'docs-tabs', 'aria-label': 'Setup
steps'} %}
{% tab label="Install" %}Run `npm install` to pull in the toolchain.{% endTab %}
{% tab label="Configure" %}Add an `aardvark.config.yaml`.{% endTab %}
{% endTabs %}
```

Source: Python

```
tabs = (
    component('aardvark', 'tab', label='Install',
              children='Run `npm install` to pull in the toolchain.')
    + component('aardvark', 'tab', label='Configure',
                children='Add an `aardvark.config.yaml`.')
)
print(component('aardvark', 'tabs', defaultValue='install',
          attr={'data-analytics': 'docs-tabs', 'aria-label': 'Setup steps'}, children=tabs))
```

Tree

A built-in tag that renders a nested file/folder explorer: collapsible folders with indentation guide lines and language-aware icons, full keyboard navigation, and per-row `defaultOpen`, `href`, `comment`, and `highlighted` options. Give a `{% file %}` a body of source code and the file becomes clickable, opening that code centered in a modal as a standard fenced code block (so the site's syntax highlighting styles it). A file with an empty body is a plain, non-clickable leaf.

Use it as `{% tree %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'tree', …)`. It is a compound block: the `{% tree %}` wrapper nests `{% folder name="…" %}` and `{% file name="…" %}` tags inside it.

Default

Wrap the structure in `{% tree %} … {% endTree %}`, nest `{% folder name="…" %} … {% endFolder %}` and `{% file name="…" %} … {% endFile %}` inside it, and put a file's source between its tags. Click a file with a body (**app.py**, **text.py**, **package.json**) to open its source in a modal with a copy button; **README.md** has no body, so it's a plain leaf. Folders collapse on click, and the whole tree is keyboard-navigable (focus it, then ↑ ↓ to move, → ← to expand/collapse, **Enter** to open a file).

Preview

Project files

src

app.py

utils

text.py

package.json

README.md

Source: Markdown

```
{% tree label="Project files" %}
{% folder name="src" defaultOpen %}
{% file name="app.py" %}
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def root():
    return {"status": "ok"}
{% endFile %}
{% folder name="utils" %}
```

```
{% file name="text.py" %}
def slugify(value: str) -> str:
    return value.strip().lower().replace(" ", "-")
{% endFile %}
{% endFolder %}
{% endFolder %}
{% file name="package.json" %}
{
  "name": "demo",
  "scripts": { "build": "vark build" }
}
{% endFile %}
{% file name="README.md" comment="start here" %}{% endFile %}
{% endTree %}
```

Source: Python

```
src = component(
    'aardvark', 'folder', name='src', defaultOpen=True,
    children=(
        component('aardvark', 'file', name='app.py',
                  children='app = FastAPI()')
        + component(
            'aardvark', 'folder', name='utils',
            children=component('aardvark', 'file', name='text.py',
                               children='def slugify(value): ...'),
        )
    ),
)
files = (
    src
    + component('aardvark', 'file', name='package.json',
                children='{ "name": "demo" }')
    + component('aardvark', 'file', name='README.md', comment='start here')
)
component('aardvark', 'tree', label='Project files', children=files)
```

Links, comments, and highlights

An `href` turns a name into a link (a bodyless file with `href` is a plain link); a `comment` adds a trailing note in muted monospace (auto-prefixed with `#`); `highlighted` tints a row to draw the eye; and `openable=false` makes a folder static (always open, no chevron).

Preview

docs

index.md

config.yaml

dist

index.html

Source: Markdown

```
{% tree %}
{% folder name="docs" defaultOpen comment="published to the site" %}
{% file name="index.md" href="/" %}{% endFile %}
{% file name="config.yaml" highlighted %}
title: My Site
theme:
  accent: indigo
{% endFile %}
{% endFolder %}
{% folder name="dist" openable=false comment="build output" %}
{% file name="index.html" %}{% endFile %}
{% endFolder %}
{% endTree %}
```

Source: Python

```
docs = component(
    'aardvark', 'folder', name='docs', defaultOpen=True,
    comment='published to the site',
    children=(
        component('aardvark', 'file', name='index.md', href='/')
        + component('aardvark', 'file', name='config.yaml', highlighted=True,
                    children='title: My Site')
    ),
)
dist = component(
    'aardvark', 'folder', name='dist', openable=False, comment='build output',
    children=component('aardvark', 'file', name='index.html'),
)
component('aardvark', 'tree', children=docs + dist)
```

With other components

A tree slots into any block — for example as one panel of a `{% tabs %}` set, paired with a prose tab that explains the layout.

Preview

Layout

Site layout

content

index.md

aardvark.config.yaml

Explanation

`content/` holds your Markdown; `aardvark.config.yaml` configures the build.

Source: Markdown

```
{% tabs defaultValue="layout" %}
{% tab label="Layout" %}
{% tree label="Site layout" %}
{% folder name="content" defaultOpen %}
{% file name="index.md" %}{% endFile %}
{% endFolder %}
{% file name="aardvark.config.yaml" comment="site config" %}{% endFile %}
{% endTree %}
{% endTab %}
{% tab label="Explanation" %}
`content/` holds your Markdown; `aardvark.config.yaml` configures the build.
{% endTab %}
{% endTabs %}
```

Source: Python

```
tree = component(
    'aardvark', 'tree', label='Site layout',
    children=(
        component('aardvark', 'folder', name='content', defaultOpen=True,
                  children=component('aardvark', 'file', name='index.md'))
        + component('aardvark', 'file', name='aardvark.config.yaml',
                    comment='site config')
    ),
)
tabs = (
    component('aardvark', 'tab', label='Layout', children=tree)
    + component('aardvark', 'tab', label='Explanation',
                children='`content/` holds your Markdown.')
)
component('aardvark', 'tabs', defaultValue='layout', children=tabs)
```

Attributes

A file's body is captured verbatim and shown as a fenced code block — the template engine does not execute inside it, so ordinary source code (even code that looks like template syntax) is safe as-is. A balanced `{% file %}` … `{% endFile %}` pair is fine too; only a lone, unbalanced end-tag needs a `{% raw %}` block to keep the engine from closing the block early.

`{% tree %}`

Attribute	Valid values	Description
`label`	string	Accessible name for the tree (sets `aria-label`); use it when a page has more than one.

`{% folder %}`

Attribute	Valid values	Description
`name`	string (required)	Folder label.
`defaultOpen`	bool flag	Start expanded instead of collapsed.
`openable`	bool (default `true`); `openable=false`	`false` → static folder: always open, no chevron, no toggle.
`href`	URL	Link the folder name (shown underlined).
`comment`	string	Trailing muted-monospace note, auto-prefixed with #.
`highlighted`	bool flag	Tint the row with the accent color.

`{% file %}`

Attribute	Valid values	Description
`name`	string (required)	File name; sets the icon and the code language.
(body)	source code	Opens in a modal on click. An empty body is a plain leaf.
`lang`	a highlight language (e.g. `bash`, `python`)	Override the language guessed from the extension.
`href`	URL	Link a bodyless file (ignored when the file has a code body — that opens the modal).
`comment`	string	Trailing muted-monospace note, auto-prefixed with #.
`highlighted`	bool flag	Tint the row with the accent color.

CSS Selectors

The tree mounts inside an island wrapper carrying `data-aardvark-island="Tree"` and renders its own class names rather than Mantine's Styles API — target the container, each row, and the file/folder labels.

```
[data-aardvark-island="Tree"]  /* the island wrapper */
.aardvark-tree                 /* the tree container */
.aardvark-tree-row             /* a single file/folder row */
.aardvark-tree-label           /* a file or folder name */
.aardvark-tree-icon            /* the leading file/folder icon */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered tree root. (Style it through the CSS parts above; set per-row options like `href`, `comment`, `highlighted`, or `lang` on each `{% file %}` / `{% folder %}`.)

Project files

src

app.py

Source: Markdown

```
{% tree label="Project files" attr={'data-analytics': 'file-tree', 'aria-label': 'Project
files'} %}
{% folder name="src" defaultOpen %}
{% file name="app.py" %}print("hello"){% endFile %}
{% endFolder %}
{% endTree %}
```

Source: Python

```
files = component('aardvark', 'folder', name='src', defaultOpen=True,
                  children=component('aardvark', 'file', name='app.py',
children='print("hello")'))
print(component('aardvark', 'tree', label='Project files',
          attr={'data-analytics': 'file-tree', 'aria-label': 'Project files'},
children=files))
```

Feedback

Built-in tags for **status and progress** — the loaders, bars, rings, gauges, and placeholders you reach for while something is happening (or has just happened). Each is a single Markdown tag with no setup, wrapping a Mantine component and exposing its full set of options.

- Loader — a spinner in three styles (oval, bars, dots).
- Navigation progress — a thin top-of-page bar you drive from JavaScript.
- Notification — a titled message card with a colored accent line.
- Progress — a progress bar, single-value or multi-segment.
- Ring progress — a circular donut ring with one or more colored sections and a center label.
- Semi-circle progress — a half-circle gauge with a center label.
- Skeleton — a shimmering placeholder for content that hasn't loaded.
- Callout — titled admonitions in four severities.

Callout

The built-in callout tag renders a titled, colored admonition with a severity color and a matching icon. `severity` is one of `success` (green), `info` (primary), `caution` (yellow), or `warning` (red); `title` is optional; and the block body is the message. Close it with `{% endCallout %}`.

Use it as `{% callout %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'callout', …)`.

Demonstrations

Severities

Set `severity` to `success`, `info`, `caution`, or `warning` — one titled example of each:

All systems go

Your changes were saved successfully.

Good to know

This setting only applies to new projects.

Double-check this

This will overwrite uncommitted local changes.

This is a destructive action

Deleting a project cannot be undone.

Source: Markdown

```
{% callout title="All systems go" severity="success" %}
Your changes were saved successfully.
{% endCallout %}

{% callout title="Good to know" severity="info" %}
This setting only applies to new projects.
{% endCallout %}

{% callout title="Double-check this" severity="caution" %}
This will overwrite uncommitted local changes.
{% endCallout %}

{% callout title="This is a destructive action" severity="warning" %}
Deleting a project cannot be undone.
{% endCallout %}
```

Source: Python

```
component('aardvark', 'callout', severity='success',
          title='All systems go',
          children='Your changes were saved successfully.')
component('aardvark', 'callout', severity='info',
          title='Good to know',
          children='This setting only applies to new projects.')
component('aardvark', 'callout', severity='caution',
          title='Double-check this',
          children='This will overwrite uncommitted local changes.')
component('aardvark', 'callout', severity='warning',
          title='This is a destructive action',
          children='Deleting a project cannot be undone.')
```

Without a title

`title` is optional — omit it for a plain icon + message:

The icon and color still convey the severity when there's no title.

Source: Markdown

```
{% callout severity="info" %}
The icon and color still convey the severity when there's no title.
{% endCallout %}
```

Source: Python

```
component('aardvark', 'callout', severity='info',
          children='The icon and color still convey the severity when there\'s no title.')
```

With other components

The block body is full Markdown, so a callout can wrap lists, links, inline code, and other tags:

Before you deploy

Check the following first:

- Your working tree is clean (`git status`).
- The `production` environment variables are set.
- You're on the right branch.

Source: Markdown

```
{% callout title="Before you deploy" severity="caution" %}
Check the following first:

- Your working tree is clean (`git status`).
```

```
- The `production` environment variables are set.
- You're on the right branch.
{% endCallout %}
```

Source: Python

```
component('aardvark', 'callout', severity='caution',
          title='Before you deploy',
          children=(
              'Check the following first:\n\n'
              '- Your working tree is clean (`git status`).\n'
              '- The `production` environment variables are set.\n'
              '- You\'re on the right branch.'))
```

Attributes

Attribute	Valid values	Description
`severity`	`success`, `info` (default), `caution`, `warning`	Severity color and matching icon. An unknown value falls back to `info`.
`title`	Any string	Optional bold heading shown above the message. Omit it for a plain icon + message.
(body)	Markdown	The message, written between `{% callout %}` and `{% endCallout %}` (`children=` from Python).

CSS Selectors

Each `callout` carries `data-aardvark-island="Callout"` on its wrapper, and the rendered Mantine Alert exposes its parts as `mantine-Alert-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Callout"] {
  /* style every callout on the page */
}

.mantine-Alert-root {
  /* the root part */
}

.mantine-Alert-title {
  /* the title part */
}

.mantine-Alert-message {
  /* the message part */
}
```

```
.mantine-Alert-icon {
  /* the icon part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

All systems go

Your changes were saved successfully.

Source: Markdown

```
{% callout title="All systems go" severity="success" attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Your changes were saved successfully.
{% endCallout %}
```

Source: Python

```
component('aardvark', 'callout', title='All systems go', severity='success',
          children='Your changes were saved successfully.', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Loader

A built-in tag for a loading spinner. It takes no body — drop it wherever something is loading and set any of the options below, or none for the default oval spinner. The three `type` styles (`oval`, `bars`, `dots`), a `size`, and a `color` cover every variant.

Use it as `{% loader %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'loader', …)`.

Demonstrations

Types

The three spinner styles — `oval` (the default), `bars`, and `dots`:

Source: Markdown

```
{% loader type='oval' %} {% loader type='bars' %} {% loader type='dots' %}
```

Source: Python

```
component('aardvark', 'loader', type='oval')
component('aardvark', 'loader', type='bars')
component('aardvark', 'loader', type='dots')
```

Size and color

`size` takes `xs`–`xl` or a number of px; `color` takes any theme color or CSS value:

Source: Markdown

```
{% loader size='sm' %}
{% loader size='lg' color='teal' %}
{% loader type='dots' size='xl' color='orange' %}
```

Source: Python

```
component('aardvark', 'loader', size='sm')
component('aardvark', 'loader', size='lg', color='teal')
component('aardvark', 'loader', type='dots', size='xl', color='orange')
```

With other components

A loader sits naturally beside other inline content — here next to a button, as an in-progress affordance:

Saving…

Source: Markdown

```
{% group %}
{% button variant="filled" %}Saving…{% endButton %}
{% loader type='dots' size='sm' %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'button', variant='filled', children='Saving…')
component('aardvark', 'loader', type='dots', size='sm')
```

Attributes

Attribute	Valid values	Description
`type`	`oval` (default), `bars`, `dots`	Spinner style.
`size`	`xs`, `sm`, `md`, `lg`, `xl`, or a number of px	Size of the spinner. Defaults to Mantine's `md`.
`color`	Any theme color (`blue`, `green`, `grape`, …) or a CSS color value	Color of the spinner.

CSS Selectors

Each `loader` carries `data-aardvark-island="Loader"` on its wrapper, and Mantine exposes its parts as `mantine-Loader-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Loader"] {
  /* style every loader on the page */
}

.mantine-Loader-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Source: Markdown

```
{% loader type='oval' attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'loader', type='oval', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

Navigation progress

A **navigation progress bar** is the thin line pinned to the very top of the page — the one that trickles forward while a route loads and snaps to full when it arrives. Unlike most components there's nothing static to drop into your prose: the bar is mounted once, then driven from JavaScript. You `start()` it, `complete()` it, or `set()` it to any value between 0 and 100, and it animates the rest.

The demo below ships as a project snippet (`snippets/NProgressDemo.jsx`) so you have something live to click. It mounts the bar and wires three buttons to the API: **Run** starts the trickle and finishes it after a beat (the way a page navigation looks), while **Start** and **Finish** expose the two ends on their own. Click any of them and watch the line at the very top of this page.

Because the snippet is a project component, it's available as a tag with no setup:

```
{% NProgressDemo %}
```

How it works

The bar is a single `<NavigationProgress />` element — mount it once (the snippet does this for the page), and from then on you move it with the imperative `nprogress` helpers rather than props:

- `nprogress.start()` — begin trickling toward the top edge (it eases forward but never reaches 100% on its own).
- `nprogress.complete()` — fill to 100% and fade out.
- `nprogress.set(value)` — jump to an exact percentage (0–100), e.g. tie it to upload progress.
- `nprogress.increment()` / `nprogress.decrement()` — nudge it by a step.
- `nprogress.reset()` — clear it back to the start with no animation.

That split — one mounted bar, an imperative API to drive it — is what makes it a snippet rather than a one-shot tag: a static `{% tag %}` can't expose `onClick` handlers that call `nprogress.start()`. The snippet owns both halves: it renders `<NavigationProgress />` and the controls that drive it. Customize it (different triggers, tie it to a real fetch, restyle the buttons) by editing `snippets/NProgressDemo.jsx`.

CSS Selectors

Every snippet instance mounts inside a wrapper carrying its island name, so you can target the demo without touching the rest of the page:

```
/* The mounted snippet wrapper (the controls live inside it). */
[data-aardvark-island="NProgressDemo"] {
  margin-block: 1rem;
}
```

The bar itself is a Mantine `Progress`, so it carries that component's stable Styles API parts — target the track with `.mantine-Progress-root` and the moving fill with `.mantine-Progress-section`:

```
/* Recolor the moving fill of the top-of-page bar. */
.mantine-Progress-section {
  background-color: var(--mantine-color-pink-6);
}
```

The bar's own layout (its fixed position at the top, the trailing glow) comes from `@mantine/nprogress`'s stylesheet, which is hashed at build time (e.g. `.m_8f2832ae` for the bar root). Those hashes change between Mantine releases, so prefer the stable `.mantine-Progress-*` parts above when you restyle it.

Injecting Attributes

`attr={...}` forwards raw HTML attributes straight onto the snippet's trigger — here the **Run** button, which is where the snippet forwards its ref. Use it to attach an inline handler, a `data-*` hook, or any attribute the component doesn't expose as a prop:

The live result — click **Run** to fire the injected `onclick` (it alerts the button's label) and run the progress bar:

Source: Markdown

```
{% NProgressDemo attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('NProgressDemo', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Attributes go on through a separate channel from React props, so they're written verbatim to the DOM node and never collide with what Mantine manages. Avoid `class`/`style` there (the framework owns those) — reach for the component's own props instead.

Notification

A built-in tag for a notification card — a titled message with a colored accent line, handy for showing a static "toast" inline in your docs. The message is the block body (or a `text` param); `title` is optional. Toggle `withBorder`, `loading`, and a decorative `withCloseButton`, and set the `color` and `radius`.

Use it as `{% notification %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'notification', …)`.

Demonstrations

Title, message, and color

The message is the block body; `title` is optional; `color` sets the accent line:

Changes saved

Your project was saved successfully.

Deploy failed

The accent line color conveys the type of message.

Source: Markdown

```
{% notification title="Changes saved" color="green" %}
Your project was saved successfully.
{% endNotification %}

{% notification title="Deploy failed" color="red" %}
The accent line color conveys the type of message.
{% endNotification %}
```

Source: Python

```
component('aardvark', 'notification', title='Changes saved', color='green',
          children='Your project was saved successfully.')
component('aardvark', 'notification', title='Deploy failed', color='red',
          children='The accent line color conveys the type of message.')
```

Border and radius

`withBorder` draws a border around the card; `radius` rounds its corners:

Heads up

A border and a yellow accent line, with extra corner rounding.

Source: Markdown

```
{% notification title="Heads up" color="yellow" withBorder=true radius="lg" %}
A border and a yellow accent line, with extra corner rounding.
{% endNotification %}
```

Source: Python

```
component('aardvark', 'notification', title='Heads up', color='yellow',
          withBorder=True, radius='lg',
          children='A border and a yellow accent line, with extra corner rounding.')
```

Loading

`loading` swaps the accent line for a spinner — use it for an in-progress message:

Uploading…

Your files are being processed.

Source: Markdown

```
{% notification title="Uploading…" loading=true %}
Your files are being processed.
{% endNotification %}
```

Source: Python

```
component('aardvark', 'notification', title='Uploading…', loading=True,
          children='Your files are being processed.')
```

Close button

`withCloseButton` shows a close button. It's decorative here (a static page has no dismiss handler), so it's off by default:

Dismissible

Pass `withCloseButton=true` to show the button.

Source: Markdown

```
{% notification title="Dismissible" color="blue" withCloseButton=true %}
Pass `withCloseButton=true` to show the button.
{% endNotification %}
```

Source: Python

```
component('aardvark', 'notification', title='Dismissible', color='blue',
```

```
            withCloseButton=True,
            children='Pass `withCloseButton=true` to show the button.')
```

With other components

A dialog and a notification combine into a **dismissable toast**: the dialog supplies the corner anchor, the trigger button, and a real dismiss (its close button), while the notification supplies the toast styling — the colored accent line and the title. Click Show notification, then dismiss it with the close button in the panel's corner.

Changes saved

Your project settings were updated.

Source: Markdown

```
{% dialog triggerLabel='Show notification' position='bottom-right' size='lg' %}
{% notification title='Changes saved' color='green' %}
Your project settings were updated.
{% endNotification %}
{% endDialog %}
```

Source: Python

```
toast = component('aardvark', 'notification', title='Changes saved', color='green',
                  children='Your project settings were updated.')
component('aardvark', 'dialog', triggerLabel='Show notification', position='bottom-right',
          size='lg', children=toast)
```

Attributes

Attribute	Valid values	Description
`title`	Any string	Optional bold heading shown above the message.
`text`	Any string	The message, when not using the block body.
`color`	Any theme color (`blue`, `green`, `red`, ...) or a CSS color value	Color of the accent line.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or a CSS value	Corner rounding of the card.
`withCloseButton`	bool (default `false`)	Show a close button. Decorative here — it has no dismiss handler on a static page.
`withBorder`	bool (default `false`)	Draw a border around the card.
`loading`	bool (default `false`)	Replace the accent line with a spinner.
(body)	Markdown	The message, written between the tags (`children=` from Python). Falls back to `text` when empty.

CSS Selectors

Each `notification` carries `data-aardvark-island="Notification"` on its wrapper, and Mantine exposes its parts as `mantine-Notification-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Notification"] {
  /* style every notification on the page */
}

.mantine-Notification-root {
  /* the root part */
}

.mantine-Notification-title {
  /* the title part */
}
```

```
.mantine-Notification-description {
  /* the description part */
}

.mantine-Notification-closeButton {
  /* the closeButton part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Changes saved

Your settings were updated.

Source: Markdown

```
{% notification title="Changes saved" color="green" attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Your settings were updated.
{% endNotification %}
```

Source: Python

```
component('aardvark', 'notification', title='Changes saved', color='green',
          children='Your settings were updated.', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Notifications

A **notifications** (toast) system you trigger from JavaScript. You mount the portal once, anywhere on the page; from then on any code — a click handler, a finished fetch, a form submit — can call `notifications.show({...})` to pop a titled message into the corner of the viewport, where it auto-dismisses. There's no component to render at the call site and no state to thread through your tree: the call is the API.

Because that's a runtime call rather than static markup, this site represents it with a small live demo snippet, `snippets/NotificationsDemo.jsx`, used as the `{% NotificationsDemo %}` tag. The snippet mounts `<Notifications />` (the portal) and a button whose click fires a toast.

```
{% NotificationsDemo %}
```

Click the button to show a notification:

Under the hood the snippet does two things — mount the portal once, then call `notifications.show(...)` from the button's click handler:

```
import { Notifications, notifications } from '@mantine/notifications';

// once, anywhere on the page:
<Notifications />

// from any handler:
notifications.show({ title: 'Saved', message: 'Your changes were saved', color: 'teal' });
```

`notifications.show` takes a `title`, a `message`, a `color`, and more (an `autoClose` timeout, an `icon`, a `loading` state, a `position`); the matching `notifications.hide`, `.update`, and `.clean` manage toasts after they're shown.

CSS Selectors

The demo mounts as a flat island, so the trigger and its wrapper carry the island marker. Target it with the `data-aardvark-island` attribute:

```
/* The whole demo island (the trigger button + the mounted portal) */
[data-aardvark-island="NotificationsDemo"] {
  /* e.g. space it from surrounding prose */
  margin-block: 0.5rem;
}
```

The toasts themselves are rendered by Mantine into a fixed-position portal, so they live **outside** the island in the DOM. Style them with Mantine's static notification classes — the portal container and each individual notification:

```
/* The portal that holds the stack of toasts */
.mantine-Notifications-root {
  /* e.g. widen the toast column */
```

```
  width: 360px;
}

/* A single toast (title, message, accent line, close button) */
.mantine-Notification-root {
  /* e.g. add a subtle shadow */
  box-shadow: var(--mantine-shadow-lg);
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the demo's **trigger element** (the button a reader clicks). Use it to add an `id`, a `data-*` hook, or a one-off inline handler without writing any JSX. Author-supplied attributes are written after React commits, so they win over React's own attribute writes.

The button below carries that extra `onclick`. Because the toast handler stays attached too, clicking it both shows the notification and runs your handler:

Source: Markdown

```
{% NotificationsDemo attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('NotificationsDemo', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Progress

A built-in tag for a progress bar. The simple form takes a single `value` (0–100) plus `color`, `size`, `radius`, and the `striped` / `animated` toggles. For a multi-segment bar, pass `sections` as a JSON array of objects — each accepting its own `value`, `color`, `striped`, and `animated`.

Use it as `{% progress %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'progress', …)`.

Demonstrations

Value

Set `value` from 0 to 100:

Source: Markdown

```
{% progress value=65 %}
```

Source: Python

```
component('aardvark', 'progress', value=65)
```

Size, radius, and color

`size` takes `xs`–`xl` or a number of px for the bar height; `radius` rounds the corners; `color` sets the fill:

Source: Markdown

```
{% progress value=30 size="sm" color="grape" %}
{% progress value=75 size="xl" radius="xl" color="cyan" %}
```

Source: Python

```
component('aardvark', 'progress', value=30, size='sm', color='grape')
component('aardvark', 'progress', value=75, size='xl', radius='xl', color='cyan')
```

Striped and animated

`striped` adds diagonal stripes; `animated` animates them (and implies `striped`):

Source: Markdown

```
{% progress value=55 striped=true %}
```

```
{% progress value=55 animated=true color="orange" %}
```

Source: Python

```
component('aardvark', 'progress', value=55, striped=True)
component('aardvark', 'progress', value=55, animated=True, color='orange')
```

Multiple segments

Pass `sections` as a JSON array — each object is one colored segment. Use it to break a bar into parts (for example, disk usage by category). Each section accepts its own `striped` and `animated` flags:

Source: Markdown

```
{% progress
sections='[{"value":35,"color":"cyan"},{"value":25,"color":"orange"},{"value":15,"color":"gr
ape"}]' size="xl" %}
{% progress
sections='[{"value":40,"color":"teal","striped":true},{"value":20,"color":"red","animated":t
rue}]' size="lg" radius="md" %}
```

Source: Python

```
component('aardvark', 'progress', size='xl', sections=(
    '[{"value":35,"color":"cyan"},'
    '{"value":25,"color":"orange"},'
    '{"value":15,"color":"grape"}]'))
component('aardvark', 'progress', size='lg', radius='md', sections=(
    '[{"value":40,"color":"teal","striped":true},'
    '{"value":20,"color":"red","animated":true}]'))
```

With other components

Label a bar with a `{% text %}` line above it, grouped in a `{% stack %}`:

Storage used — 70%

Source: Markdown

```
{% stack gap="xs" %}
{% text size="sm" fw="500" %}Storage used — 70%{% endText %}
{% progress value=70 color="teal" %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'text', size='sm', fw='500', children='Storage used — 70%')
```

```
component('aardvark', 'progress', value=70, color='teal')
```

Attributes

Attribute	Valid values	Description
`value`	A number 0–100	Fill percentage (simple, single-segment form).
`color`	Any theme color or a CSS color value	Bar color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`, or a number of px	Bar height. Defaults to Mantine's `md`.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or a CSS value	Corner rounding.
`striped`	bool (default `false`)	Add diagonal stripes.
`animated`	bool (default `false`)	Animate the stripes (implies `striped`).
`sections`	A JSON array of `{value, color, striped?, animated?}` objects	Multi-segment bar. Each object is one colored segment; mutually exclusive with `value`. Invalid JSON degrades to a build-time HTML comment.

CSS Selectors

Each `progress` carries `data-aardvark-island="Progress"` on its wrapper, and Mantine exposes its parts as `mantine-Progress-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Progress"] {
  /* style every progress on the page */
}

.mantine-Progress-root {
  /* the root part */
}

.mantine-Progress-section {
  /* the section part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Source: Markdown

```
{% progress value=65 attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'progress', value=65, attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

Ring progress

A built-in tag for a circular (donut) progress ring. Pass `sections` as a JSON array of `{value, color}` objects — one arc per object — and optionally set a center `label`, the `size` and `thickness` in px, and `roundCaps` for rounded arc ends.

Use it as `{% ringprogress %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'ringprogress', …)`.

Demonstrations

Single section

The minimal form — one section and a center label:

40%

Source: Markdown

```
{% ringprogress sections='[{"value":40,"color":"cyan"}]' label="40%" %}
```

Source: Python

```
component('aardvark', 'ringprogress',
          sections='[{"value":40,"color":"cyan"}]', label='40%')
```

Multiple sections

Each object in `sections` is one colored arc; the values stack around the ring. `size` sets the ring's width and height in px:

70%

Source: Markdown

```
{% ringprogress
sections='[{"value":40,"color":"cyan"},{"value":15,"color":"orange"},{"value":15,"color":"gr
ape"}]' label="70%" size=140 %}
```

Source: Python

```
component('aardvark', 'ringprogress', label='70%', size=140, sections=(
    '[{"value":40,"color":"cyan"},'
    '{"value":15,"color":"orange"},'
    '{"value":15,"color":"grape"}]'))
```

Thickness and rounded caps

`thickness` sets the arc width in px; `roundCaps` rounds the ends of each arc:

65%

Source: Markdown

```
{% ringprogress sections='[{"value":65,"color":"teal"}]' label="65%" thickness=16
roundCaps=true size=140 %}
```

Source: Python

```
component('aardvark', 'ringprogress',
          sections='[{"value":65,"color":"teal"}]',
          label='65%', thickness=16, roundCaps=True, size=140)
```

With other components

Pair a ring with a caption underneath, grouped in a centered `{% stack %}`:

82%

Tests passing

Source: Markdown

```
{% stack align="center" gap="xs" %}
{% ringprogress sections='[{"value":82,"color":"teal"}]' label="82%" size=120 %}
{% text size="sm" c="dimmed" %}Tests passing{% endText %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'ringprogress',
          sections='[{"value":82,"color":"teal"}]', label='82%', size=120)
component('aardvark', 'text', size='sm', c='dimmed', children='Tests passing')
```

Attributes

Attribute	Valid values	Description
`sections`	A JSON array of `{value, color}` objects	**Required.** One arc per object. Invalid JSON degrades to a build-time HTML comment.
`size`	A number of px	Width and height of the ring. Defaults to Mantine's 120.
`thickness`	A number of px	Arc thickness. Defaults to Mantine's ~12.
`roundCaps`	bool (default `false`)	Round the ends of each arc.
`label`	Any string	Text shown in the center of the ring.

CSS Selectors

Each `ringprogress` carries `data-aardvark-island="RingProgress"` on its wrapper, and Mantine exposes its parts as `mantine-RingProgress-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="RingProgress"] {
  /* style every ringprogress on the page */
}

.mantine-RingProgress-root {
  /* the root part */
}

.mantine-RingProgress-svg {
  /* the svg part */
}

.mantine-RingProgress-curve {
  /* the curve part */
}

.mantine-RingProgress-label {
  /* the label part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

40%

Source: Markdown

```
{% ringprogress sections='[{"value":40,"color":"cyan"}]' label="40%" attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'ringprogress',
          sections='[{"value":40,"color":"cyan"}]', label='40%', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Semi-circle progress

A built-in tag for a half-circle (gauge) progress indicator. Set `value` (0–100) and, optionally, a `label`, the gauge `size` and `thickness`, an `orientation` (`up` or `down`), a `fillDirection`, the `labelPosition`, and the filled / empty segment colors.

Use it as `{% semicircleprogress %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'semicircleprogress', …)`.

Demonstrations

Value and label

The minimal form — a `value` and an optional `label`:

65%

Source: Markdown

```
{% semicircleprogress value=65 label="65%" %}
```

Source: Python

```
component('aardvark', 'semicircleprogress', value=65, label='65%')
```

Colors and thickness

`filledSegmentColor` colors the filled arc, `emptySegmentColor` the empty track, and `thickness` sets the arc width in px:

80%

Source: Markdown

```
{% semicircleprogress value=80 label="80%" filledSegmentColor="teal"
emptySegmentColor="gray" thickness=16 %}
```

Source: Python

```
component('aardvark', 'semicircleprogress', value=80, label='80%',
          filledSegmentColor='teal', emptySegmentColor='gray', thickness=16)
```

Orientation, fill direction, and label position

`orientation='down'` flips the gauge; `fillDirection='right-to-left'` fills the other way; `labelPosition='center'` moves the label into the arc:

45%

Source: Markdown

```
{% semicircleprogress value=45 label="45%" orientation="down" fillDirection="right-to-left"
labelPosition="center" filledSegmentColor="grape" %}
```

Source: Python

```
component('aardvark', 'semicircleprogress', value=45, label='45%',
          orientation='down', fillDirection='right-to-left',
          labelPosition='center', filledSegmentColor='grape')
```

With other components

Caption a gauge with a `{% text %}` heading, grouped in a centered `{% stack %}`:

CPU load

72%

Source: Markdown

```
{% stack align="center" gap="0" %}
{% text size="sm" fw="500" %}CPU load{% endText %}
{% semicircleprogress value=72 label="72%" size=160 filledSegmentColor="orange" %}
{% endStack %}
```

Source: Python

```
component('aardvark', 'text', size='sm', fw='500', children='CPU load')
component('aardvark', 'semicircleprogress', value=72, label='72%',
          size=160, filledSegmentColor='orange')
```

Attributes

Attribute	Valid values	Description
`value`	A number 0–100	**Required.** Fill percentage.
`size`	A number of px	Diameter of the gauge. Defaults to 200.
`thickness`	A number of px	Arc thickness. Defaults to 12.
`orientation`	`up` (default), `down`	Which way the half-circle opens.
`fillDirection`	`left-to-right` (default), `right-to-left`	Direction the arc fills.
`label`	Any string	Text shown inside the gauge.
`labelPosition`	`bottom` (default), `center`	Where the label sits.
`filledSegmentColor`	Any theme color or a CSS color value	Color of the filled arc.
`emptySegmentColor`	Any theme color or a CSS color value	Color of the empty track.

CSS Selectors

Each `semicircleprogress` carries `data-aardvark-island="SemiCircleProgress"` on its wrapper, and Mantine exposes its parts as `mantine-SemiCircleProgress-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="SemiCircleProgress"] {
  /* style every semicircleprogress on the page */
}

.mantine-SemiCircleProgress-root {
  /* the root part */
}

.mantine-SemiCircleProgress-filledSegment {
  /* the filledSegment part */
}

.mantine-SemiCircleProgress-emptySegment {
  /* the emptySegment part */
```

```
}

.mantine-SemiCircleProgress-label {
  /* the label part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

65%

Source: Markdown

```
{% semicircleprogress value=65 label="65%" attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'semicircleprogress', value=65, label='65%', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Skeleton

A built-in tag for a loading placeholder — a shimmering grey block that stands in for content that hasn't loaded yet. Set a `height` (and optionally a `width`), `radius`, or `circle` for an avatar shape. Wrap real content as the block body and reveal it with `visible=false`; `animate=false` stops the shimmer.

Use it as `{% skeleton %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'skeleton', …)`.

Demonstrations

Lines

Set a `height` and an optional `width` to mock up text lines:

Source: Markdown

```
{% skeleton height='1rem' %}
{% skeleton height='1rem' width='70%' %}
{% skeleton height='1rem' width='40%' %}
```

Source: Python

```
component('aardvark', 'skeleton', height='1rem')
component('aardvark', 'skeleton', height='1rem', width='70%')
component('aardvark', 'skeleton', height='1rem', width='40%')
```

Circle and radius

`circle` makes an avatar-shaped placeholder (width and radius track the height); `radius` rounds a rectangular one:

Source: Markdown

```
{% group %}
{% skeleton circle=true height='48' %}
{% skeleton height='48' width='200' radius='md' %}
{% endGroup %}
```

Source: Python

```
component('aardvark', 'skeleton', circle=True, height='48')
component('aardvark', 'skeleton', height='48', width='200', radius='md')
```

Static (no shimmer)

`animate=false` keeps a plain grey placeholder with no shimmer animation:

Source: Markdown

```
{% skeleton height='1rem' width='60%' animate=false %}
```

Source: Python

```
component('aardvark', 'skeleton', height='1rem', width='60%', animate=False)
```

Wrapping content

Wrap real content as the block body and reveal it with `visible=false` — the skeleton overlays the content while `visible` is `true`:

This content shows through once `visible=false`.

Source: Markdown

```
{% skeleton visible=false %}
This content shows through once `visible=false`.
{% endSkeleton %}
```

Source: Python

```
component('aardvark', 'skeleton', visible=False,
          children='This content shows through once `visible=false`.')
```

With other components

Compose a few skeletons into a card-style placeholder — a circle avatar beside two lines, all inside a `{% paper %}`:

Source: Markdown

```
{% paper withBorder=true p="md" radius="md" %}
{% group %}
{% skeleton circle=true height='40' %}
{% stack gap="xs" %}
{% skeleton height='0.8rem' width='160' %}
{% skeleton height='0.8rem' width='100' %}
{% endStack %}
{% endGroup %}
{% endPaper %}
```

Source: Python

```
component('aardvark', 'skeleton', circle=True, height='40')
component('aardvark', 'skeleton', height='0.8rem', width='160')
component('aardvark', 'skeleton', height='0.8rem', width='100')
```

Attributes

Attribute	Valid values	Description
`height`	A number (px → rem) or any CSS value	Block height.
`width`	A number or CSS value	Block width. Defaults to `100%`; ignored when `circle` is set.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	Corner rounding of a rectangular placeholder.
`circle`	bool (default `false`)	Render a circle (width and radius track the height) — good for an avatar placeholder.
`visible`	bool (default `true`)	When `false`, show the wrapped block body instead of the shimmer.
`animate`	bool (default `true`)	Toggle the shimmer animation.
(body)	Markdown	Content the skeleton overlays; revealed when `visible=false` (`children=` from Python).

CSS Selectors

Each `skeleton` carries `data-aardvark-island="Skeleton"` on its wrapper, and Mantine exposes its parts as `mantine-Skeleton-*` classes — target either to style it from your theme CSS.

```
[data-aardvark-island="Skeleton"] {
  /* style every skeleton on the page */
}

.mantine-Skeleton-root {
  /* the root part */
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Source: Markdown

```
{% skeleton height='1rem' attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% endSkeleton %}
```

Source: Python

```
component('aardvark', 'skeleton', height='1rem', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

Overlays

UI that layers over the page. The **panels** — modal, drawer, dialog — ship with a built-in trigger button and open/close wiring, so they work on a static page out of the box (click the trigger, then close via the button, the overlay, or Escape). The **floating** overlays anchor a transient surface to a target using Mantine's floating engine (placement, arrows, offsets, flip-on-collision). Each is a single **built-in** tag that forwards the full Mantine prop surface.

Panels

- Modal — a centered dialog over a dimming overlay, with a trigger button.
- Drawer — a panel that slides in from any edge, with a trigger button.
- Dialog — a small corner-anchored panel (no backdrop), with a trigger button.
- Affix — pins content to a fixed spot in the viewport as the page scrolls.
- Overlay — a dimming or gradient veil over a box, for spotlighting or disabled states.
- LoadingOverlay — a spinner veil over an area that's loading.
- FloatingWindow — a draggable titled panel you can grab and move.
- Spotlight — a Cmd+K command palette of searchable actions, opened from JavaScript.

Managers

- Modals manager — `@mantine/modals`, the imperative modal manager you open from JavaScript (`modals.openConfirmModal`), shown as a live demo.

Floating

- Tooltip — a small floating label shown on hover or focus of a target.
- Popover — a click-toggled floating panel whose body is full Markdown.
- HoverCard — a richer Tooltip that opens on hover, with a Markdown body and delays.
- Menu — a dropdown of actions and links, with section labels and dividers.
- FloatingIndicator — the low-level sliding-highlight primitive behind SegmentedControl, Tabs, and Stepper.

Motion

- Transition — fade / slide / scale content in and out as it mounts.

Affix

A **built-in** tag that pins its content to a fixed spot in the viewport — it stays put as the page scrolls, which makes it ideal for a “back to top” button, a chat launcher, or any floating action. Pick a corner with `position` and the pixel distance from those edges with `offset`; `zIndex` controls stacking. The block body is the pinned content.

Use it as `{% affix %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'affix', …)`.

Demonstrations

A bottom-right affix (the default corner). It pins to the viewport, so the button below stays in the corner as you scroll the page:

Back to top

Source: Markdown

```
{% affix position='bottom-right' offset=24 %}
{% button text='Back to top' url='#' variant='filled' %}
{% endAffix %}
```

Source: Python

```
component(
    'aardvark', 'affix',
    position='bottom-right',
    offset=24,
    children=component('aardvark', 'button', text='Back to top', url='#', variant='filled'),
)
```

Anchored to other corners and edges

`position` takes any corner (`bottom-right`, `bottom-left`, `top-right`, `top-left`). A single token like `top` or `left` is completed into a corner using the default `bottom`/`right` (so `top` → top-right, `left` → bottom-left). A larger `offset` pushes it further from the named edges, and `zIndex` lifts it above other fixed elements:

Top-left action

Source: Markdown

```
{% affix position='top-left' offset=80 zIndex=300 %}
{% button text='Top-left action' variant='light' color='grape' %}
{% endAffix %}
```

Source: Python

```
component(
    'aardvark', 'affix',
    position='top-left',
    offset=80,
    zIndex=300,
    children=component('aardvark', 'button', text='Top-left action', variant='light',
color='grape'),
)
```

With other components

The block body is ordinary Markdown, so any tag can be pinned. Here a card holds a small stack of actions in the bottom-right corner:

Help

Source: Markdown

```
{% affix position='bottom-right' offset=24 zIndex=300 %}
{% card shadow='md' padding='sm' radius='md' withBorder=true %}
{% button text='Help' variant='light' size='xs' fullWidth=true %}
{% endCard %}
{% endAffix %}
```

Source: Python

```
inner = component('aardvark', 'button', text='Help', variant='light', size='xs',
fullWidth=True)
card = component('aardvark', 'card', shadow='md', padding='sm', radius='md',
withBorder=True, children=inner)
component('aardvark', 'affix', position='bottom-right', offset=24, zIndex=300,
children=card)
```

Because an affix is pinned to the viewport rather than rendered in the page flow, it sits over your content — keep it small and out of the way of the main reading column.

Attributes

Attribute	Values	Description
`position`	`bottom-right` (default), `bottom-left`, `top-right`, `top-left` (a single token like `top` or `left` is completed to a corner with the default `bottom`/`right`)	Corner to pin the content to.
`offset`	integer (px), default `20`	Distance from each named edge.
`zIndex`	integer	Stacking order, for sitting above or below other fixed elements. Omitted by default so Mantine's default applies.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Affix"]`, or through the Mantine Styles API classes:

```
/* Every rendered Affix carries this island marker */
[data-aardvark-island="Affix"] { }

/* Mantine Styles API classes */
.mantine-Affix-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Back to top

Source: Markdown

```
{% affix position='bottom-right' offset=24 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% button text='Back to top' url='#' variant='filled' %}
{% endAffix %}
```

Source: Python

```
component('aardvark', 'affix', position='bottom-right', offset=24,
          children=component('aardvark', 'button', text='Back to top', url='#',
variant='filled'),
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Dialog

A **built-in** tag for a dialog — a small, fixed panel anchored to a corner of the viewport, with no backdrop (the rest of the page stays interactive). It ships with a trigger button and the open/close state wired up. Since there's no overlay to click, dismiss it with its own close button. Anchor it with `position`, size and shape it with `size` / `radius` / `withBorder`, and tune the open/close with the props below. It starts closed and opens from its trigger button.

Use it as `{% dialog %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'dialog', …)`.

Demonstrations

The trigger button opens the dialog; the block body is its content:

A quick, non-blocking message in the corner.

Source: Markdown

```
{% dialog triggerLabel='Show the dialog' position='bottom-right' %}
A quick, non-blocking message in the corner.
{% endDialog %}
```

Source: Python

```
component(
    'aardvark', 'dialog',
    triggerLabel='Show the dialog',
    position='bottom-right',
    children='A quick, non-blocking message in the corner.',
)
```

Bordered, top-left, larger

`position` anchors to any corner or edge; `size` widens the panel, `radius` rounds it, and `withBorder` draws an outline:

A bordered dialog anchored to the top-left corner.

Source: Markdown

```
{% dialog triggerLabel='Top-left dialog' position='top-left' size='lg' radius='md'
withBorder=true %}
A bordered dialog anchored to the top-left corner.
{% endDialog %}
```

Source: Python

```
component(
    'aardvark', 'dialog',
    triggerLabel='Top-left dialog',
    position='top-left',
    size='lg',
    radius='md',
    withBorder=True,
    children='A bordered dialog anchored to the top-left corner.',
)
```

A tuned pop-in transition

`transition` / `transitionDuration` tune how the dialog appears from its corner. Because a Dialog has no backdrop, its close button is the only way to dismiss it — so it always stays available:

A corner dialog that pops in. Dismiss it with the close button.

Source: Markdown

```
{% dialog triggerLabel='Open the dialog' position='bottom-left' transition='pop'
transitionDuration=200 %}
A corner dialog that pops in. Dismiss it with the close button.
{% endDialog %}
```

Source: Python

```
component(
    'aardvark', 'dialog',
    triggerLabel='Open the dialog',
    position='bottom-left',
    transition='pop',
    transitionDuration=200,
    children='A corner dialog that pops in. Dismiss it with the close button.',
)
```

With other components

A dialog and a notification combine into a **dismissable toast**: the dialog supplies the corner anchor, the trigger button, and a real dismiss (its close button), while the notification supplies the toast styling — the colored accent line and the title. Click Show notification, then dismiss it with the close button in the panel's corner.

Changes saved

Your project settings were updated.

Source: Markdown

```
{% dialog triggerLabel='Show notification' position='bottom-right' size='lg' %}
{% notification title='Changes saved' color='green' %}
Your project settings were updated.
{% endNotification %}
{% endDialog %}
```

Source: Python

```
toast = component('aardvark', 'notification', title='Changes saved', color='green',
                  children='Your project settings were updated.')
component('aardvark', 'dialog', triggerLabel='Show notification', position='bottom-right',
          size='lg', children=toast)
```

Attributes

Attribute	Values	Description
`triggerLabel`	string, default `Open dialog`	Text on the opener button.
`opened`	bool, default `false`	Start the dialog already open on load. Usually leave this off so it opens from the trigger button.
`position`	`bottom-right` (default), `bottom-left`, `top-right`, `top-left`, or a single edge	Corner or edge to anchor to.
`size`	`xs`–`xl`, a number of px, or a percentage	Panel width.
`radius`	`xs`–`xl` or a number	Corner radius.
`withBorder`	bool, default `false`	Draw a border around the panel.
`withCloseButton`	bool, default `true`	Show the close button. It’s the dialog’s only dismissal (there’s no backdrop or Escape), so if you turn it off the build re-enables it and warns.
`transition`	`pop`, `fade`, `slide-up`, …	Open/close transition.
`transitionDuration`	integer (ms)	Transition length.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Dialog"]`, or through the Mantine Styles API classes. Its parts render only while the dialog is shown. The relevant classes:

```
/* Every rendered Dialog carries this island marker */
[data-aardvark-island="Dialog"] { }

/* Mantine Styles API classes */
.mantine-Dialog-root { }
.mantine-Dialog-closeButton { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered overlay. Open the dialog and click it: the injected `onclick` reads the overlay's text and alerts it.

Click this dialog to run the injected handler.

Source: Markdown

```
{% dialog triggerLabel='Show the dialog' position='bottom-right' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Click this dialog to run the injected handler.
{% endDialog %}
```

Source: Python

```
component('aardvark', 'dialog', triggerLabel='Show the dialog', position='bottom-right',
          children='Click this dialog to run the injected handler.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Drawer

A **built-in** tag for a drawer — a panel that slides in from an edge of the screen over a dimming overlay. Like Modal, it ships with a trigger button and the open/close state wired up, so it works on a static page: click the trigger to open, then close via the close button, the overlay, or Escape. Pick the edge it slides from with `position`, set the header with `title`, and tune size, padding, and the overlay/transition with the props below. The drawer starts closed and opens from its trigger button, so the page never loads with the panel in the reader's way.

Use it as `{% drawer %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'drawer', …)`.

Demonstrations

The trigger button opens the drawer; the block body is its content:

Drawer content — any **Markdown** works here.

Source: Markdown

```
{% drawer title='Navigation' triggerLabel='Open the drawer' position='right' %}
Drawer content — any **Markdown** works here.
{% endDrawer %}
```

Source: Python

```
component(
    'aardvark', 'drawer',
    title='Navigation',
    triggerLabel='Open the drawer',
    position='right',
    children='Drawer content — any **Markdown** works here.',
)
```

A detached drawer from the bottom

`position` picks the edge; `offset` floats the drawer off the edges, `radius` rounds its corners, and `padding` / `size` size the panel:

A detached drawer that slides up from the bottom.

Source: Markdown

```
{% drawer title='Filters' triggerLabel='Open from bottom' position='bottom' offset=16
radius='md' padding='lg' size='md' %}
A detached drawer that slides up from the bottom.
{% endDrawer %}
```

Source: Python

```
component(
    'aardvark', 'drawer',
    title='Filters',
    triggerLabel='Open from bottom',
    position='bottom',
    offset=16,
    radius='md',
    padding='lg',
    size='md',
    children='A detached drawer that slides up from the bottom.',
)
```

A tuned overlay and a deliberate close

This drawer darkens and blurs the backdrop (`overlayBackgroundOpacity`, `overlayBlur`) and tunes the slide (`transition`, `transitionDuration`). It also sets `closeOnClickOutside=false` so a stray click on the dimmed backdrop won't dismiss it — you close it deliberately with the header button or the Escape key:

Your cart slides in over a blurred backdrop. Close it with the header button or Escape — clicking the backdrop won't dismiss it.

Source: Markdown

```
{% drawer title='Cart' triggerLabel='Open the cart' position='left'
overlayBackgroundOpacity=0.6 overlayBlur=2 transition='slide-right' transitionDuration=250
closeOnClickOutside=false %}
Your cart slides in over a blurred backdrop. Close it with the header button or
Escape — clicking the backdrop won't dismiss it.
{% endDrawer %}
```

Source: Python

```
component(
    'aardvark', 'drawer',
    title='Cart',
    triggerLabel='Open the cart',
    position='left',
    overlayBackgroundOpacity=0.6,
    overlayBlur=2,
    transition='slide-right',
    transitionDuration=250,
    closeOnClickOutside=False,
    children=(
        'Your cart slides in over a blurred backdrop. Close it with the header '
        'button or Escape — clicking the backdrop won\'t dismiss it.'
    ),
```

```
)
```

With other components

The body is ordinary Markdown, so a drawer can hold a navigation list or a form. Here a button closes out an applied-filters panel:

Choose the facets to narrow the list.

Apply filters

Source: Markdown

```
{% drawer title='Filters' triggerLabel='Filter results' position='right' size='sm' %}
Choose the facets to narrow the list.

{% button text='Apply filters' variant='filled' fullWidth=true %}
{% endDrawer %}
```

Source: Python

```
apply = component('aardvark', 'button', text='Apply filters', variant='filled',
fullWidth=True)
component('aardvark', 'drawer', title='Filters', triggerLabel='Filter results',
position='right', size='sm',
          children='Choose the facets to narrow the list.\n\n' + apply)
```

Attributes

Attribute	Values	Description
`title`	string	Header text shown beside the close button.
`triggerLabel`	string, default `Open drawer`	Text on the opener button.
`opened`	bool, default `false`	Start the drawer already open on load. Usually leave this off so the page opens to the trigger button rather than a panel covering the content.
`position`	`left` (default), `right`, `top`, `bottom`	Edge the drawer slides from.
`size`	`xs`–`xl`, a number of px, or a percentage	Panel width (or height for top/bottom).
`offset`	integer (px)	Gap between the drawer and the viewport edges (a detached drawer).
`radius`	`xs`–`xl` or a number	Corner radius.
`padding`	`xs`–`xl` or a number	Inner padding.
`withCloseButton`	bool, default `true`	Show the header close button.
`closeOnClickOutside`	bool, default `true`	Close when the overlay is clicked.
`closeOnEscape`	bool, default `true`	Close on the Escape key.
`overlayBackgroundOpacity`	float `0`–`1`	Overlay opacity.
`overlayBlur`	float (px)	Backdrop blur behind the drawer.

`transition`	`slide-right`, `slide-left`, `fade`, …	Open/close transition.
`transitionDuration`	integer (ms)	Transition length.

A drawer can always be closed: if you set `withCloseButton`, `closeOnClickOutside`, and `closeOnEscape` all to `false` there'd be no way out, so the build re-enables the close button (and prints a warning). A drawer can never trap the reader.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Drawer"]`, or through the Mantine Styles API classes. The drawer mounts into a portal and its parts exist only while it is open. The relevant classes:

```
/* Every rendered Drawer carries this island marker */
[data-aardvark-island="Drawer"] { }

/* Mantine Styles API classes */
.mantine-Drawer-root { }
.mantine-Drawer-overlay { }
.mantine-Drawer-content { }
.mantine-Drawer-header { }
.mantine-Drawer-title { }
.mantine-Drawer-body { }
.mantine-Drawer-close { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered overlay. Open the drawer and click anywhere in it: the injected `onclick` reads the overlay's text and alerts it.

Click anywhere in this drawer to run the injected handler.

Source: Markdown

```
{% drawer title='Navigation' triggerLabel='Open the drawer' position='right'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Click anywhere in this drawer to run the injected handler.
{% endDrawer %}
```

Source: Python

```
component('aardvark', 'drawer', title='Navigation', triggerLabel='Open the drawer',
position='right',
          children='Click anywhere in this drawer to run the injected handler.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

FloatingIndicator

`FloatingIndicator` is a **low-level positioning primitive**: it animates a single highlight box from one target element to another, given a root ref and a target ref. It has no behavior of its own — it is the engine Mantine uses inside **SegmentedControl**, **Tabs**, and **Stepper** to slide their active highlight. Because the primitive needs live element refs (which a build-time tag can't supply), this tag renders a small self-contained **demo**: a row of buttons whose active highlight slides on click, set by `labels` (a comma-separated list).

Use it as `{% floatingindicator %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'floatingindicator', …)`.

Demonstration

Click the segments — the highlight box slides between them. The sliding box is the `FloatingIndicator`; the row and click handling around it are the wiring that a real control (SegmentedControl, Tabs) provides for you.

Source: Markdown

```
{% floatingindicator labels='Day, Week, Month' %}
```

Source: Python

```
component('aardvark', 'floatingindicator', labels=['Day', 'Week', 'Month'])
```

The default `labels` is `One, Two, Three`, so the bare tag still renders three segments:

Source: Markdown

```
{% floatingindicator %}
```

Source: Python

```
component('aardvark', 'floatingindicator')
```

With other components

The primitive is meant to live inside a real control. In practice you reach for a component that already wires the indicator for you — for example Aardvark's built-in Tabs, which slides an underline indicator on switch. The demo below shows the standalone primitive sitting beside a short note rendered by the Text tag:

The highlight above is the same primitive Tabs uses internally.

Source: Markdown

```
{% floatingindicator labels='Code, Preview, Diff' %}

{% text size='sm' c='dimmed' %}The highlight above is the same primitive Tabs uses
internally.{% endText %}
```

Source: Python

```
component('aardvark', 'floatingindicator', labels=['Code', 'Preview', 'Diff'])
component('aardvark', 'text', size='sm', c='dimmed',
          children='The highlight above is the same primitive Tabs uses internally.')
```

Attributes

`FloatingIndicator` has no build-time-supplyable props of its own — it is a primitive that animates a highlight between two element refs, so the only input is the demo's button labels. Style and behavior come from the control that hosts it (see Tabs for a production-ready animated indicator).

Attribute	Valid values	Description
`labels`	Comma-separated list of strings	Button labels for the demo row whose active highlight slides on click. Defaults to `One, Two, Three`.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="FloatingIndicator"]`, or through the Mantine Styles API classes. The Mantine root is the sliding highlight box. The relevant classes:

```
/* Every rendered FloatingIndicator carries this island marker */
[data-aardvark-island="FloatingIndicator"] { }

/* Mantine Styles API classes */
.mantine-FloatingIndicator-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% floatingindicator labels='Day, Week, Month' attr={'onclick': '''
```

```
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'floatingindicator', labels=['Day', 'Week', 'Month'],
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

FloatingWindow

A **built-in** tag for a draggable floating panel — a titled window that opens **over the page** and floats above your content; grab its header to move it around the viewport. It's a handy container for a help bubble, a mini-map, or any side panel the reader can reposition. A **trigger button** opens it (set its text with `triggerLabel`), then set the header text with `title`, size it with `width`, and toggle the close button and border with `withCloseButton` and `withBorder`.

Use it as `{% floatingwindow %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'floatingwindow', …)`.

Demonstrations

Click the trigger to open a titled window that floats over the page — grab its header and drag it around:

Tips

Drag me by the title bar to move this window around the page.

Source: Markdown

```
{% floatingwindow triggerLabel='Show tips' title='Tips' width=320 %}
**Drag me** by the title bar to move this window around the page.
{% endFloatingwindow %}
```

Source: Python

```
component(
    'aardvark', 'floatingwindow',
    triggerLabel='Show tips',
    title='Tips',
    width=320,
    children='**Drag me** by the title bar to move this window around the page.',
)
```

A borderless window without a close button

`shadow` and `radius` restyle the card; `withCloseButton=false` and `withBorder=false` strip the chrome for a lighter panel:

Note

A lighter floating panel — no border, no close button.

Source: Markdown

```
{% floatingwindow triggerLabel='Open note' title='Note' width=280 shadow='md' radius='lg'
```

```
withCloseButton=false withBorder=false %}
A lighter floating panel — no border, no close button.
{% endFloatingwindow %}
```

Source: Python

```
component(
    'aardvark', 'floatingwindow',
    triggerLabel='Open note',
    title='Note',
    width=280,
    shadow='md',
    radius='lg',
    withCloseButton=False,
    withBorder=False,
    children='A lighter floating panel — no border, no close button.',
)
```

With other components

The body is ordinary Markdown, so a window can hold richer content — here a button inside a draggable help panel:

Need help?

Reach support without leaving the page.

Contact us

Source: Markdown

```
{% floatingwindow triggerLabel='Need help?' title='Need help?' width=300 shadow='xl' %}
Reach support without leaving the page.

{% button text='Contact us' variant='light' fullWidth=true %}
{% endFloatingwindow %}
```

Source: Python

```
btn = component('aardvark', 'button', text='Contact us', variant='light', fullWidth=True)
component('aardvark', 'floatingwindow', triggerLabel='Need help?', title='Need help?',
          width=300, shadow='xl',
          children='Reach support without leaving the page.\n\n' + btn)
```

Attributes

Attribute	Values	Description
`triggerLabel`	string, default `Open window`	Text of the button that opens the window.
`title`	string	Header text shown in the draggable title bar.
`width`	integer (px), default `320`	Window width.
`shadow`	`xs`–`xl` (default `xl`)	Drop shadow.
`radius`	`xs`–`xl` or a number (default `md`)	Corner radius.
`withCloseButton`	bool, default `true`	Show the close button in the header.
`withBorder`	bool, default `true`	Draw a border around the window.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="FloatingWindow"]`, or through the Mantine Styles API classes. The window mounts on the client and exists only while it is open. The relevant classes:

```
/* Every rendered FloatingWindow carries this island marker */
[data-aardvark-island="FloatingWindow"] { }

/* Mantine Styles API classes */
.mantine-FloatingWindow-root { }

/* The demo adds these to the title bar */
.aardvark-fw-bar { }
.aardvark-fw-close { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Tips

Drag me by the title bar to move this window around the page.

Source: Markdown

```
{% floatingwindow triggerLabel='Show tips' title='Tips' width=320 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
**Drag me** by the title bar to move this window around the page.
{% endFloatingwindow %}
```

Source: Python

```
component('aardvark', 'floatingwindow', triggerLabel='Show tips', title='Tips', width=320,
          children='**Drag me** by the title bar to move this window around the page.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

HoverCard

A floating card that opens when you **hover** its trigger — think of it as a Tooltip whose body is full **Markdown** rather than a plain label. `target` is the inline trigger text; the block body is the card content. The card reveals on hover after the page hydrates, so in a static screenshot you'll see only the trigger link; hover it in a live browser to open the card.

Use it as `{% hovercard %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'hovercard', …)`.

Demonstrations

Basic

`target` is the inline trigger; everything between the tags is the card body. Hover the trigger to reveal it.

Hover

Aardvark — a static site generator.

Markdown in, fast static site out.

to see the card.

Source: Markdown

```
Hover {% hovercard target='@aardvark' width='280' shadow='md' withArrow=true %}
**Aardvark** — a static site generator.

Markdown in, fast static site out.
{% endHovercard %} to see the card.
```

Source: Python

```
component('aardvark', 'hovercard', target='@aardvark', width='280',
          shadow='md', withArrow=True,
          children='**Aardvark** — a static site generator.\n\n'
                   'Markdown in, fast static site out.')
```

Position and delays

`position` anchors the card on any edge; `openDelay` / `closeDelay` tune how eagerly it appears and lingers, so you can move the pointer onto the card without it snapping shut.

Hover

Opens after a beat, and lingers so you can move onto it.

here.

Source: Markdown

```
Hover {% hovercard target='slowly' position='top' openDelay=300 closeDelay=200 width='240'
withArrow=true %}
Opens after a beat, and lingers so you can move onto it.
{% endHovercard %} here.
```

Source: Python

```
component('aardvark', 'hovercard', target='slowly', position='top',
          openDelay=300, closeDelay=200, width='240', withArrow=True,
          children='Opens after a beat, and lingers so you can move onto it.')
```

Radius, offset, and arrow size

`radius` rounds the card, `offset` sets the gap to the trigger, and `arrowSize` sizes the pointer that `withArrow` draws.

Hover

A roomier card with a larger arrow and more breathing room.

for the full look.

Source: Markdown

```
Hover {% hovercard target='details' width='260' radius='lg' offset=12 withArrow=true
arrowSize=10 shadow='lg' %}
A roomier card with a larger arrow and more breathing room.
{% endHovercard %} for the full look.
```

Source: Python

```
component('aardvark', 'hovercard', target='details', width='260',
          radius='lg', offset=12, withArrow=True, arrowSize=10, shadow='lg',
          children='A roomier card with a larger arrow and more breathing room.')
```

With other components

Because the body is full Markdown, it can host other tags. Here the card holds a Badge and a link:

Hover

[Operational]

All systems normal. See the status page.

for details.

Source: Markdown

```
Hover {% hovercard target='status' width='240' shadow='md' withArrow=true %}
{% badge color='green' variant='light' %}Operational{% endBadge %}

All systems normal. See the [status page](/).
{% endHovercard %} for details.
```

Source: Python

```
component('aardvark', 'hovercard', target='status', width='240',
          shadow='md', withArrow=True,
          children=component('aardvark', 'badge', color='green',
                             variant='light', children='Operational')
                   + '\n\nAll systems normal. See the [status page](/).')
```

HoverCard vs. Tooltip vs. Popover

- **Tooltip** — a short plain-text label, on hover.
- **HoverCard** — a richer Markdown body, on hover (this tag).
- **Popover** — a Markdown panel, on click.

Attributes

Omit any attribute to take its Mantine default. The card body is the block body (or, from Python, `children`).

Attribute	Valid values	Description
`target`	string	The inline trigger text. Defaults to `Hover me`.
`position`	`bottom` (default), `top`, `left`, `right`, plus the `-start` / `-end` variants	Edge the card anchors to.
`width`	integer (pixels), or `target` to match the trigger	Card width.
`shadow`	`xs`–`xl`	Drop shadow on the card.
`withArrow`	bare flag → `true`	Draw a pointer at the anchored edge.

`arrowSize`	integer (pixels)	Arrow size.
`offset`	integer (pixels)	Gap between the trigger and the card.
`radius`	`xs`–`xl` or a number	Card corner radius.
`openDelay`	integer (milliseconds)	Delay before opening on hover.
`closeDelay`	integer (milliseconds)	Delay before closing after the pointer leaves.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="HoverCard"]`, or through the Mantine Styles API classes. The card mounts into a portal and its parts exist only while it is shown. The relevant classes:

```
/* Every rendered HoverCard carries this island marker */
[data-aardvark-island="HoverCard"] { }

/* Mantine Styles API classes */
.mantine-HoverCard-root { }
.mantine-HoverCard-dropdown { }
.mantine-HoverCard-arrow { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered card. Hover the target to show the card, then click it: the injected `onclick` reads the card's text and alerts it.

Hover

Click this card to run the injected handler.

to see the card.

Source: Markdown

```
Hover {% hovercard target='@aardvark' width='280' shadow='md' withArrow=true
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
```

```
'''} %}
Click this card to run the injected handler.
{% endHovercard %} to see the card.
```

Source: Python

```
component('aardvark', 'hovercard', target='@aardvark', width='280',
          shadow='md', withArrow=True,
          children='Click this card to run the injected handler.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

LoadingOverlay

A **built-in** tag for a loading veil — a dimming layer with a centered spinner over a box of content, for an area that's fetching or processing. The tag wraps the block body in a positioned box and lays the veil over it, so it renders right here on the page. Toggle the veil with `visible`, tune it with the `overlay*` props, and style the spinner with the `loader*` props.

Use it as `{% loadingoverlay %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'loadingoverlay', …)`.

Demonstrations

The covered content is the block body; the veil is on by default (`visible=true`):

This content sits under the loading veil while it's `visible`.

Source: Markdown

```
{% loadingoverlay %}
This content sits under the loading veil while it's `visible`.
{% endLoadingoverlay %}
```

Source: Python

```
component(
    'aardvark', 'loadingoverlay',
    children='This content sits under the loading veil while it is visible.',
)
```

A blurred veil with a custom spinner

`overlayBlur` frosts the content behind the veil; `loaderType`, `loaderColor`, and `loaderSize` restyle the spinner:

The veil blurs the content behind it, with a grape "bars" spinner.

Source: Markdown

```
{% loadingoverlay overlayBlur=2 overlayBackgroundOpacity=0.55 loaderType='bars'
loaderColor='grape' loaderSize='lg' %}
The veil blurs the content behind it, with a grape "bars" spinner.
{% endLoadingoverlay %}
```

Source: Python

```
component(
    'aardvark', 'loadingoverlay',
    overlayBlur=2,
```

```
    overlayBackgroundOpacity=0.55,
    loaderType='bars',
    loaderColor='grape',
    loaderSize='lg',
    children='The veil blurs the content behind it, with a grape bars spinner.',
)
```

Content revealed (veil hidden)

Set `visible=false` to lift the veil and show the content underneath. Combined with the `overlayRadius` and `zIndex` props, this is how you'd drive the overlay from loading to ready in a real app:

The veil is hidden, so this content reads normally.

Source: Markdown

```
{% loadingoverlay visible=false overlayRadius='md' %}
The veil is hidden, so this content reads normally.
{% endLoadingoverlay %}
```

Source: Python

```
component(
    'aardvark', 'loadingoverlay',
    visible=False,
    overlayRadius='md',
    children='The veil is hidden, so this content reads normally.',
)
```

With other components

The body is ordinary Markdown, so the veil can sit over richer content — here a card being "loaded":

Account summary

Numbers refresh while the veil is up.

Source: Markdown

```
{% loadingoverlay overlayBlur=1 loaderType='dots' %}
{% card shadow='sm' padding='lg' radius='md' withBorder=true %}
### Account summary
```

```
Numbers refresh while the veil is up.
{% endCard %}
{% endLoadingoverlay %}
```

Source: Python

```
body = component('aardvark', 'card', shadow='sm', padding='lg', radius='md',
withBorder=True,
                 children='### Account summary\n\nNumbers refresh while the veil is up.')
component('aardvark', 'loadingoverlay', overlayBlur=1, loaderType='dots', children=body)
```

Attributes

Attribute	Values	Description
`visible`	bool, default `true`	Show the veil + spinner. Set `false` to reveal the covered content.
`overlayBackgroundOpacity`	float `0`–`1`	Veil opacity.
`overlayBlur`	float (px)	Backdrop blur behind the veil.
`overlayRadius`	`xs`–`xl` or a number	Veil corner radius.
`loaderType`	`oval` (default), `bars`, `dots`	Spinner style.
`loaderSize`	`xs`–`xl` or a number	Spinner size.
`loaderColor`	any theme color	Spinner color.
`zIndex`	integer	Stacking order of the veil.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="LoadingOverlay"]`, or through the Mantine Styles API classes:

```
/* Every rendered LoadingOverlay carries this island marker */
[data-aardvark-island="LoadingOverlay"] { }

/* Mantine Styles API classes */
.mantine-LoadingOverlay-root { }
.mantine-LoadingOverlay-overlay { }
```

```
.mantine-LoadingOverlay-loader { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

This content sits under the loading veil while it's `visible`.

Source: Markdown

```
{% loadingoverlay attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
This content sits under the loading veil while it's `visible`.
{% endLoadingoverlay %}
```

Source: Python

```
component(
    'aardvark', 'loadingoverlay',
    children='This content sits under the loading veil while it is `visible`.',
    attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
)
```

Menu

A dropdown menu of actions and links anchored to a trigger button. `label` is the trigger button; `items` is a JSON array describing the entries — actions, links, section headings, and dividers. The dropdown opens on click (or hover) after the page hydrates, so in a static screenshot you'll see only the trigger button; click it in a live browser to open the menu.

Use it as `{% menu %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'menu', …)`.

Each entry in `items` is a small JSON object:

- **An item** — a `label`, plus optional `href` (renders the item as a link), `color` (e.g. `red` for a destructive action), and `disabled`.
- **A section heading** — `{"section": "…"}`, a non-clickable group label.
- **A divider** — `{"divider": true}`, a horizontal rule.

Demonstrations

Sections and dividers

A `section` entry adds a non-clickable group label; a `divider` entry draws a rule between groups.

Actions

Source: Markdown

```
{% menu label='Actions' items='[
  {"section": "Account"},
  {"label": "Profile", "href": "/"},
  {"label": "Settings", "href": "/"},
  {"divider": true},
  {"label": "Sign out", "color": "red"}
]' %}
```

Source: Python

```
component('aardvark', 'menu', label='Actions', items='''[
  {"section": "Account"},
  {"label": "Profile", "href": "/"},
  {"label": "Settings", "href": "/"},
  {"divider": true},
  {"label": "Sign out", "color": "red"}
]''')
```

Links, colors, and disabled items

An `href` turns an item into a link; `color` tints a destructive action; `disabled` greys one out. `targetVariant` restyles the trigger, `position` anchors the dropdown, and `withArrow` points it at the button.

Repo

Source: Markdown

```
{% menu label='Repo' targetVariant='light' position='bottom-start' withArrow=true items='[
  {"label": "View on GitHub", "href": "https://github.com/"},
  {"label": "Download .zip", "href": "/"},
  {"divider": true},
  {"label": "Archive", "disabled": true},
  {"label": "Delete", "color": "red"}
]' %}
```

Source: Python

```
component('aardvark', 'menu', label='Repo', targetVariant='light',
          position='bottom-start', withArrow=True, items='''[
  {"label": "View on GitHub", "href": "https://github.com/"},
  {"label": "Download .zip", "href": "/"},
  {"divider": true},
  {"label": "Archive", "disabled": true},
  {"label": "Delete", "color": "red"}
]''')
```

Open on hover

Set `trigger='hover'` to open the menu on hover instead of click. `width`, `shadow`, `offset`, and `radius` size and style the dropdown.

Hover

Source: Markdown

```
{% menu label='Hover' trigger='hover' width=180 shadow='lg' offset=4 radius='md'
items='[{"label": "One"}, {"label": "Two"}]' %}
```

Source: Python

```
component('aardvark', 'menu', label='Hover', trigger='hover', width=180,
          shadow='lg', offset=4, radius='md',
          items='[{"label": "One"}, {"label": "Two"}]')
```

Keep the menu open after a click

By default the menu closes once an item is chosen. Set `closeOnItemClick=false` to keep it open — handy for a menu of toggles.

View

Source: Markdown

```
{% menu label='View' closeOnItemClick=false items='[
  {"section": "Toggles"},
  {"label": "Show grid"},
  {"label": "Show rulers"},
  {"label": "Snap to pixels"}
]' %}
```

Source: Python

```
component('aardvark', 'menu', label='View', closeOnItemClick=False, items='''[
  {"section": "Toggles"},
  {"label": "Show grid"},
  {"label": "Show rulers"},
  {"label": "Snap to pixels"}
]''')
```

With other components

Because `items` is plain data, a Python caller can build the menu from a loop — for example, mapping a list of pages to link entries — and render it alongside a Text heading:

Documentation

Sections

Source: Markdown

```
{% text fw='600' %}Documentation{% endText %}

{% menu label='Sections' position='bottom-start' items='[
  {"label": "Getting started", "href": "/"},
  {"label": "Components", "href": "/"},
  {"label": "Configuration", "href": "/"}
]' %}
```

Source: Python

```
import json

pages = [
    ('Getting started', '/'),
```

```
    ('Components', '/'),
    ('Configuration', '/'),
]
items = json.dumps([{'label': name, 'href': url} for name, url in pages])

component('aardvark', 'text', fw='600', children='Documentation')
component('aardvark', 'menu', label='Sections',
          position='bottom-start', items=items)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`label`	string	The trigger button's text. Defaults to `Menu`.
`items`	JSON array string	The menu entries. Each is an item object (`label`, plus optional `href`, `color`, `disabled`), a section object (`{"section": "…"}`), or a divider object (`{"divider": true}`).
`position`	`bottom-start`, `bottom`, `top`, `right-start`, … (plus `-start` / `-end` variants)	Edge the dropdown anchors to.
`width`	integer (pixels)	Dropdown width.
`shadow`	`xs`–`xl`	Drop shadow on the dropdown.
`trigger`	`click` (default) / `hover`	How the menu opens.
`withArrow`	bare flag → `true`	Draw a pointer at the anchored edge.
`offset`	integer (pixels)	Gap between the trigger and the dropdown.
`radius`	`xs`–`xl` or a number	Dropdown corner radius.
`targetVariant`	`filled`, `light`, `outline`, `subtle`, `default`, …	The trigger button's Mantine variant.
`closeOnItemClick`	`true` (default) / `false`	Close the menu after an item is chosen. Set `false` to keep it open.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Menu"]`, or through the Mantine Styles API classes. The dropdown mounts into a portal and its parts exist only while the menu is open. The relevant classes:

```
/* Every rendered Menu carries this island marker */
[data-aardvark-island="Menu"] { }

/* Mantine Styles API classes */
.mantine-Menu-dropdown { }
.mantine-Menu-item { }
.mantine-Menu-label { }
.mantine-Menu-divider { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered dropdown. Open the menu and click an item: the injected `onclick` reads the dropdown's text and alerts it.

Actions

Source: Markdown

```
{% menu label='Actions' items='[
  {"section": "Account"},
  {"label": "Profile"},
  {"label": "Sign out", "color": "red"}
]' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'menu', label='Actions', items='''[
  {"section": "Account"},
  {"label": "Profile"},
  {"label": "Sign out", "color": "red"}
]''', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Modal

A **built-in** tag for a modal dialog — a centered panel over a dimming overlay. It ships with a trigger button and the open/close state wired up, so it works on a static page with no setup: click the trigger to open, then close via the close button, the overlay, or Escape. Set `triggerLabel` for the opener text, `title` for the header, and tune size, layout, and the overlay/transition with the props below. The modal starts closed and opens from its trigger button, so the page never loads with a panel covering the content.

Use it as `{% modal %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'modal', …)`.

Demonstrations

The trigger button opens the modal; the block body is the modal content:

This is the modal body — any **Markdown** works here.

Source: Markdown

```
{% modal title='Welcome' triggerLabel='Open the modal' %}
This is the modal body — any **Markdown** works here.
{% endModal %}
```

Source: Python

```
component(
    'aardvark', 'modal',
    title='Welcome',
    triggerLabel='Open the modal',
    children='This is the modal body — any **Markdown** works here.',
)
```

A larger, centered modal

`size` widens the panel, `centered` pins it to the vertical middle, and `radius` / `padding` / `overlayBlur` restyle the panel and its backdrop:

A bigger, vertically-centered modal with a blurred backdrop.

Source: Markdown

```
{% modal title='Settings' triggerLabel='Open settings' size='lg' centered=true radius='md'
padding='xl' overlayBlur=3 %}
A bigger, vertically-centered modal with a blurred backdrop.
{% endModal %}
```

Source: Python

```
component(
    'aardvark', 'modal',
    title='Settings',
    triggerLabel='Open settings',
    size='lg',
    centered=True,
    radius='md',
    padding='xl',
    overlayBlur=3,
    children='A bigger, vertically-centered modal with a blurred backdrop.',
)
```

A full-screen modal

`fullScreen` expands the modal to fill the whole viewport — good for an immersive form or an onboarding step. It still opens from its trigger and closes with the header button or the Escape key (a full-screen panel has no surrounding backdrop to click). `transition` / `transitionDuration` tune the open animation:

A full-screen modal. Close it with the header button or the Escape key.

Source: Markdown

```
{% modal title='Onboarding' triggerLabel='Start onboarding' fullScreen=true
transition='fade' transitionDuration=200 %}
A full-screen modal. Close it with the header button or the Escape key.
{% endModal %}
```

Source: Python

```
component(
    'aardvark', 'modal',
    title='Onboarding',
    triggerLabel='Start onboarding',
    fullScreen=True,
    transition='fade',
    transitionDuration=200,
    children='A full-screen modal. Close it with the header button or the Escape key.',
)
```

With other components

The body is ordinary Markdown, so a modal can hold a whole form. Here a button confirms an action and `overlayBackgroundOpacity` darkens the backdrop:

This permanently removes the project and all of its data.

Yes, delete it

Source: Markdown

```
{% modal title='Delete project?' triggerLabel='Delete' centered=true
overlayBackgroundOpacity=0.7 %}
This permanently removes the project and all of its data.

{% button text='Yes, delete it' color='red' variant='filled' %}
{% endModal %}
```

Source: Python

```
confirm = component('aardvark', 'button', text='Yes, delete it', color='red',
variant='filled')
component('aardvark', 'modal', title='Delete project?', triggerLabel='Delete',
centered=True,
          overlayBackgroundOpacity=0.7,
          children='This permanently removes the project and all of its data.\n\n' +
confirm)
```

Attributes

Attribute	Values	Description
`title`	string	Header text shown beside the close button.
`triggerLabel`	string, default `Open modal`	Text on the opener button.
`opened`	bool, default `false`	Start the modal already open on load. Usually leave this off so the page opens to the trigger button rather than a panel over the content.
`size`	`xs`–`xl`, a number of px, `auto`, or `100%`	Panel width.
`radius`	`xs`–`xl` or a number	Corner radius.
`padding`	`xs`–`xl` or a number	Inner padding.
`centered`	bool, default `false`	Vertically center the modal (off → it sits near the top).
`fullScreen`	bool, default `false`	Expand to fill the whole viewport.
`withCloseButton`	bool, default `true`	Show the header close button.
`closeOnClickOutside`	bool, default `true`	Close when the overlay is clicked.
`closeOnEscape`	bool, default `true`	Close on the Escape key.
`overlayBackgroundOpacity`	float `0`–`1`	Overlay opacity.
`overlayBlur`	float (px)	Backdrop blur behind the modal.
`transition`	`fade`, `pop`, `slide-up`, …	Open/close transition.

`transitionDuration`	integer (ms)	Transition length.

A modal can always be closed: if you set `withCloseButton`, `closeOnClickOutside`, and `closeOnEscape` all to `false` there'd be no way out, so the build re-enables the close button (and prints a warning). A modal can never trap the reader.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Modal"]`, or through the Mantine Styles API classes. The modal mounts into a portal and its parts exist only while it is open. The relevant classes:

```
/* Every rendered Modal carries this island marker */
[data-aardvark-island="Modal"] { }

/* Mantine Styles API classes */
.mantine-Modal-root { }
.mantine-Modal-overlay { }
.mantine-Modal-content { }
.mantine-Modal-header { }
.mantine-Modal-title { }
.mantine-Modal-body { }
.mantine-Modal-close { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered overlay. Open the modal and click anywhere in it: the injected `onclick` reads the overlay's text and alerts it.

Click anywhere in this modal to run the injected handler.

Source: Markdown

```
{% modal title='Welcome' triggerLabel='Open the modal' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Click anywhere in this modal to run the injected handler.
{% endModal %}
```

Source: Python

```
component(
    'aardvark', 'modal',
    title='Welcome',
    triggerLabel='Open the modal',
    children='Click anywhere in this modal to run the injected handler.',
```

```
    attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
)
```

Modals manager

`@mantine/modals` is an **imperative modal manager**: instead of placing a dialog in the page, you call a function — `modals.openConfirmModal()`, `modals.open()`, `modals.openContextModal()` — from an event handler, and the manager mounts, stacks, and tears down the dialog for you. A single `<ModalsProvider>` near the root owns the open-modal state and renders each dialog into a portal; closing one (via its action, the close button, the backdrop, or Escape) pops it off the stack. Because that whole surface lives at runtime — a function you call, not a static element — there is no single tag that captures it. So this page demonstrates it with a small, self-contained island: a button that opens a confirm dialog through the manager.

The `{% ModalsDemo %}` tag below comes from `snippets/ModalsDemo.jsx`, which wraps a Mantine `Button` in `<ModalsProvider>` and, on click, calls `modals.openConfirmModal({ title, children, labels })`. Defining the snippet is what makes it addressable as a tag; see the component libraries guide for the broader snippet-to-tag mechanism.

Demonstration

Click the button to open a confirm dialog. It has a title, a body, and the two labelled action buttons (`OK` / `Cancel`); dismiss it from either button, the backdrop, or Escape.

Source: Markdown

```
{% ModalsDemo %}
```

Source: snippet

```
import { forwardRef } from 'react';
import { Button } from '@mantine/core';
import { ModalsProvider, modals } from '@mantine/modals';

const ModalsDemo = forwardRef(function ModalsDemo(props, ref) {
  return (
    <ModalsProvider>
      <Button
        ref={ref}
        {...props}
        onClick={() =>
          modals.openConfirmModal({
            title: 'Confirm',
            children: 'Proceed?',
            labels: { confirm: 'OK', cancel: 'Cancel' },
          })
        }
      >
        Open confirm dialog
      </Button>
    </ModalsProvider>
```

```
  );
});

export default ModalsDemo;
```

ModalsProvider and modals.openConfirmModal

Two pieces make the manager work:

- **`ModalsProvider`** is the host. Mount it once around the part of the tree that opens dialogs (in the demo, around the trigger button); it holds the open-modal state and portals each dialog above the page. Nothing renders until you ask it to, so the page loads with no panel over the content.
- **`modals.openConfirmModal(payload)`** pushes a ready-made confirm dialog onto the manager and returns its id. The payload's `title` and `children` fill the header and body, and `labels: { confirm, cancel }` names the two action buttons; pass `onConfirm` / `onCancel` handlers to react to the choice. Sibling calls — `modals.open()` for an arbitrary body, `modals.openContextModal()` for a pre-registered modal, and `modals.closeAll()` — drive the same manager.

Because these are runtime calls rather than props, they live in the snippet's JavaScript, not in the Markdown — the tag just renders the wired-up trigger.

CSS Selectors

The island's mount point carries `data-aardvark-island="ModalsDemo"`, so you can target the trigger (and anything the snippet renders at its root) without touching the snippet:

```
/* The demo's mount point — the trigger button lives here */
[data-aardvark-island="ModalsDemo"] {
  display: inline-block;
}
```

The dialog itself is rendered by the manager into a portal, so its Mantine parts are addressable by their standard Styles-API class names — but only **once a modal is open** (the portal is empty until then):

```
/* Present only while a dialog is open */
.mantine-Modal-root {
  z-index: 1000;
}

.mantine-Modal-content {
  border-radius: var(--mantine-radius-md);
}
```

Injecting Attributes

The snippet forwards its `ref` to the trigger button, so an `attr={...}` map lands as raw DOM attributes on that button (the `attr` channel applies them through the forwarded ref, separate from React props). Use it to attach an inline handler or any HTML attribute the React prop surface doesn't cover:

(`ModalsDemo` is a custom snippet wrapping the modals **manager** — not one of the built-in overlay tags — so it keeps `attr` on its trigger button. The built-in `{% modal %}` tag, by contrast, now attaches `attr` to the opened overlay.)

The live result — clicking it both runs the injected `onclick` and opens the dialog:

Source: Markdown

```
{% ModalsDemo attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('ModalsDemo', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Overlay

A **built-in** tag for a dimming veil over a box — for spotlighting content, a hover scrim, or a disabled state. The tag wraps the veil in a sized box so it's visible right here on the page, and the block body renders centered on top of the veil. Tune the veil with `color`, `backgroundOpacity`, `blur`, and `radius`, or swap the flat color for a `gradientFrom`/`gradientTo`/`gradientDeg` linear gradient.

Use it as `{% overlay %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'overlay', …)`.

Demonstrations

A flat dimming veil — set the `color` and `backgroundOpacity` (`0`–`1`); the body sits centered above it:

Spotlighted content

Source: Markdown

```
{% overlay color='#000' backgroundOpacity=0.6 %}
{% text c='white' fw='700' %}Spotlighted content{% endText %}
{% endOverlay %}
```

Source: Python

```
label = component('aardvark', 'text', c='white', fw='700', children='Spotlighted content')
component('aardvark', 'overlay', color='#000', backgroundOpacity=0.6, children=label)
```

A blurred veil

`blur` frosts whatever sits behind the veil (in px); `radius` rounds the box:

Frosted

Source: Markdown

```
{% overlay color='#000' backgroundOpacity=0.35 blur=4 radius='md' %}
{% text c='white' fw='700' %}Frosted{% endText %}
{% endOverlay %}
```

Source: Python

```
label = component('aardvark', 'text', c='white', fw='700', children='Frosted')
component('aardvark', 'overlay', color='#000', backgroundOpacity=0.35, blur=4, radius='md',
children=label)
```

A gradient veil

Pass `gradientFrom` / `gradientTo` / `gradientDeg` for a fade instead of a flat color — its start/end colors and angle:

Bottom-up fade

Source: Markdown

```
{% overlay gradientFrom='rgba(0,0,0,0.8)' gradientTo='rgba(0,0,0,0)' gradientDeg=0 %}
{% text c='white' fw='700' %}Bottom-up fade{% endText %}
{% endOverlay %}
```

Source: Python

```
label = component('aardvark', 'text', c='white', fw='700', children='Bottom-up fade')
component(
    'aardvark', 'overlay',
    gradientFrom='rgba(0,0,0,0.8)',
    gradientTo='rgba(0,0,0,0)',
    gradientDeg=0,
    children=label,
)
```

With other components

The body is ordinary Markdown, so any tag can sit on the veil. Here a button reads as a call to action through a dimmed scrim:

Unlock this section

Source: Markdown

```
{% overlay color='#000' backgroundOpacity=0.5 radius='md' %}
{% button text='Unlock this section' variant='filled' color='grape' %}
{% endOverlay %}
```

Source: Python

```
cta = component('aardvark', 'button', text='Unlock this section', variant='filled',
color='grape')
component('aardvark', 'overlay', color='#000', backgroundOpacity=0.5, radius='md',
children=cta)
```

Attributes

Attribute	Values	Description
`color`	any CSS color	Veil color.
`backgroundOpacity`	float `0`–`1`	Veil opacity.
`blur`	float (px)	Backdrop blur behind the veil.
`radius`	`xs`–`xl` or a number	Corner radius of the box.
`zIndex`	integer	Stacking order.
`gradientFrom`	any CSS color	Start color of a linear-gradient veil (used in place of a flat `color`).
`gradientTo`	any CSS color	End color of the gradient.
`gradientDeg`	integer (degrees)	Angle of the gradient.

Setting any of `gradientFrom` / `gradientTo` / `gradientDeg` composes a `linear-gradient` veil and overrides the flat `color`. Unset gradient endpoints default to a dark-to-transparent fade at `90°`.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Overlay"]`, or through the Mantine Styles API classes:

```
/* Every rendered Overlay carries this island marker */
[data-aardvark-island="Overlay"] { }

/* Mantine Styles API classes */
.mantine-Overlay-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Spotlighted content

Source: Markdown

```
{% overlay color='#000' backgroundOpacity=0.6 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% text c='white' fw='700' %}Spotlighted content{% endText %}
{% endOverlay %}
```

Source: Python

```
label = component('aardvark', 'text', c='white', fw='700', children='Spotlighted content')
print(component('aardvark', 'overlay', color='#000', backgroundOpacity=0.6, children=label,
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}))
```

Popover

A floating panel that toggles open when you **click** its trigger. `target` is the trigger button's label; the block body is the panel content and is ordinary **Markdown** — headings, lists, links, even nested tags. The panel toggles on click after the page hydrates; pass `opened` to force it open for docs and screenshots.

Use it as `{% popover %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'popover', …)`.

Demonstrations

Basic

`target` is the button label; everything between the tags is the dropdown body. Click the button to toggle it.

Signed in as aardvark

- Profile
- Settings

Source: Markdown

```
{% popover target='Account' position='bottom' shadow='md' withArrow=true %}
**Signed in as** aardvark

- [Profile](/)
- [Settings](/)
{% endPopover %}
```

Source: Python

```
component('aardvark', 'popover', target='Account', position='bottom',
          shadow='md', withArrow=True,
          children='**Signed in as** aardvark\n\n- [Profile](/)\n- [Settings](/)')
```

Width, arrow, and trigger variant

`width='target'` makes the panel exactly as wide as its trigger; `withArrow` points at the button; `targetVariant` restyles the trigger; and `arrowSize`, `offset`, and `radius` tune the rest of the look.

The panel matches the trigger's width.

Source: Markdown

```
{% popover target='More' width='target' withArrow=true arrowSize=8 offset=6 radius='md'
position='bottom-start' targetVariant='light' %}
The panel matches the trigger's width.
{% endPopover %}
```

Source: Python

```
component('aardvark', 'popover', target='More', width='target',
          withArrow=True, arrowSize=8, offset=6, radius='md',
          position='bottom-start', targetVariant='light',
          children="The panel matches the trigger's width.")
```

Focus and click-outside behavior

`trapFocus` keeps keyboard focus inside the panel (useful when it holds a form), and `closeOnClickOutside=false` keeps the panel open when you click away.

Keyboard focus is trapped here, and clicking outside won't dismiss it.

Source: Markdown

```
{% popover target='Filter' trapFocus=true closeOnClickOutside=false shadow='md' width='240'
%}
Keyboard focus is trapped here, and clicking outside won't dismiss it.
{% endPopover %}
```

Source: Python

```
component('aardvark', 'popover', target='Filter', trapFocus=True,
          closeOnClickOutside=False, shadow='md', width='240',
          children="Keyboard focus is trapped here, and clicking outside "
                   "won't dismiss it.")
```

Force it open

`opened=true` renders the dropdown open so you can see (and screenshot) its contents without clicking:

This panel is pinned open for the docs.

Source: Markdown

```
{% popover target='Pinned' opened=true shadow='lg' width='260' %}
This panel is pinned open for the docs.
{% endPopover %}
```

Source: Python

```
component('aardvark', 'popover', target='Pinned', opened=True,
```

```
          shadow='lg', width='260',
          children='This panel is pinned open for the docs.')
```

With other components

The Markdown body can host other tags. Here the panel holds a Badge and a Text note, pinned open so you can see them:

[Passing]

Last run 3 minutes ago on `main`.

Source: Markdown

```
{% popover target='Build' shadow='md' width='260' withArrow=true %}
{% badge color='green' variant='light' %}Passing{% endBadge %}

{% text size='sm' c='dimmed' %}Last run 3 minutes ago on `main`.{% endText %}
{% endPopover %}
```

Source: Python

```
component('aardvark', 'popover', target='Build', shadow='md', width='260',
          withArrow=True,
          children=component('aardvark', 'badge', color='green',
                             variant='light', children='Passing')
                   + '\n\n'
                   + component('aardvark', 'text', size='sm', c='dimmed',
                               children='Last run 3 minutes ago on `main`.'))
```

Popover vs. HoverCard vs. Menu

- **Popover** — a Markdown panel, on click (this tag).
- **HoverCard** — a Markdown card, on hover.
- **Menu** — a list of actions and links, on click.

Attributes

Omit any attribute to take its Mantine default. The dropdown body is the block body (or, from Python, `children`).

Attribute	Valid values	Description

`target`	string	The trigger button's label. Defaults to `Toggle`.
`position`	`bottom` (default), `top`, `left`, `right`, plus the `-start` / `-end` variants	Edge the dropdown anchors to.
`width`	integer (pixels), or `target` to match the trigger	Dropdown width.
`shadow`	`xs–xl`	Drop shadow on the dropdown.
`withArrow`	bare flag → `true`	Draw a pointer at the anchored edge.
`arrowSize`	integer (pixels)	Arrow size.
`offset`	integer (pixels)	Gap between the trigger and the dropdown.
`radius`	`xs–xl` or a number	Dropdown corner radius.
`trapFocus`	bare flag → `true`	Keep keyboard focus inside the dropdown while open.
`closeOnClickOutside`	`true` (default) / `false`	Close when clicking away. Set `false` to keep it open.
`targetVariant`	`filled`, `light`, `outline`, `subtle`, `default`, …	The trigger button's Mantine variant.
`opened`	bare flag → `true`	Force the dropdown open — handy for documentation and screenshots.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Popover"]`, or through the Mantine Styles API classes. The dropdown mounts into a portal and its parts exist only while the popover is shown. The relevant classes:

```
/* Every rendered Popover carries this island marker */
[data-aardvark-island="Popover"] { }

/* Mantine Styles API classes */
.mantine-Popover-dropdown { }
```

```
.mantine-Popover-arrow { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered dropdown. Click **Account** to open the popover, then click inside it: the injected `onclick` reads the dropdown's text and alerts it.

Click here to run the injected handler.

Source: Markdown

```
{% popover target='Account' position='bottom' shadow='md' withArrow=true attr={'onclick':
'''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Click here to run the injected handler.
{% endPopover %}
```

Source: Python

```
component('aardvark', 'popover', target='Account', position='bottom',
          shadow='md', withArrow=True, children='Click here to run the injected handler.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Spotlight

A command palette — the Cmd+K overlay you see in editors and dashboards. It is a searchable list of actions that floats over the page; you type to filter, arrow through the matches, and press Enter to run one. It renders no visible chrome of its own and starts closed, so it never covers the page on load. You open it imperatively from JavaScript with `spotlight.open()` — wired here to a trigger button — or with its registered keyboard shortcut.

Because the open call is imperative rather than a static prop, this page ships it as a small self-contained snippet, `{% SpotlightDemo %}`, that renders the palette plus a button whose click calls `spotlight.open()`.

Demonstrations

Click the button to open the palette, then type to filter the actions, arrow to a match, and press Enter (or click) to run it. Escape closes it.

Source: Markdown

```
{% SpotlightDemo %}
```

The actions array and spotlight.open()

The palette is driven by an **actions array** — the searchable command list. Each entry is an object with an `id`, a `label`, an optional `description`, and an `onClick` handler that runs when the action is triggered:

```
const actions = [
  { id: 'home', label: 'Home', description: 'Go to the home page',
    onClick: () => { /* navigate, run a command, … */ } },
  { id: 'search', label: 'Search the site', keywords: ['find', 'lookup'],
    onClick: () => { /* … */ } },
];
```

The default filter matches the query against each action's `label`, `description`, and `keywords`. Triggering an action runs its `onClick` and closes the palette.

Opening is imperative: any code can call `spotlight.open()` (and `spotlight.close()` / `spotlight.toggle()`). Here the trigger button does it on click:

```
import { Spotlight, spotlight } from '@mantine/spotlight';
import { Button } from '@mantine/core';

<>
  <Spotlight actions={actions} nothingFound="Nothing found…" />
  <Button onClick={spotlight.open}>Open command palette</Button>
</>
```

The full snippet lives at `snippets/SpotlightDemo.jsx`; because it sits in `snippets/`, it is automatically invocable as `{% SpotlightDemo %}`.

CSS Selectors

The snippet mounts as an island whose wrapper carries `data-aardvark-island="SpotlightDemo"`, so you can target the rendered demo as a whole:

```
/* The whole SpotlightDemo island (trigger button + the palette portal) */
[data-aardvark-island="SpotlightDemo"] {
  display: inline-block;
}
```

The palette itself renders into a portal at the end of `<body>`, outside the island wrapper, and uses Mantine's per-part static classes. The two you reach for most are the palette root and its search input:

```
/* The palette overlay's root element */
.mantine-Spotlight-root {
  border-radius: 12px;
}

/* The search input at the top of the palette */
.mantine-Spotlight-search {
  font-size: 1rem;
}
```

Injecting Attributes

The snippet forwards its `ref` to the trigger button, so an `attr={…}` map lands on that button's DOM node — the same raw-attribute channel the built-in button uses for `onclick`. Pass any HTML attribute (including inline event handlers):

Here is that live — clicking the button both opens the palette and runs the injected `onclick`:

Source: Markdown

```
{% SpotlightDemo attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('SpotlightDemo', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
```

```
alert(value);
'''})
```

Tooltip

A floating label that appears when you hover or focus a single target. The target is the block body; `label` is the floating text. It ships with Aardvark, so a tooltip is a single tag pair with no setup. The tooltip reveals on hover or focus after the page hydrates — pass `opened` to force it visible in docs and screenshots.

Use it as `{% tooltip %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'tooltip', …)`.

Demonstrations

Basic

Hover the target to reveal the label.

Saved 2 minutes ago

Hover me

Source: Markdown

```
{% tooltip label='Saved 2 minutes ago' %}Hover me{% endTooltip %}
```

Source: Python

```
component('aardvark', 'tooltip', label='Saved 2 minutes ago', children='Hover me')
```

Position, color, and arrow

`position` anchors the tooltip on any edge (`top` / `bottom` / `left` / `right`, plus the `-start` / `-end` variants), `color` tints its background, and `withArrow` draws a pointer. Each tooltip below is pinned open so you can see all three at once:

Above, with an arrow

Top + arrow

On the right

Right + color

Below the target

Bottom

Source: Markdown

```
{% tooltip label='Above, with an arrow' position='top' withArrow=true %}Top + arrow{%
endTooltip %}
{% tooltip label='On the right' position='right' color='grape' withArrow=true %}Right +
color{% endTooltip %}
{% tooltip label='Below the target' position='bottom' color='red' %}Bottom{% endTooltip %}
```

Source: Python

```
component('aardvark', 'tooltip', label='Above, with an arrow',
          position='top', withArrow=True, children='Top + arrow')
component('aardvark', 'tooltip', label='On the right',
          position='right', color='grape', withArrow=True, children='Right + color')
component('aardvark', 'tooltip', label='Below the target',
          position='bottom', color='red', children='Bottom')
```

Multiline

For longer labels, turn on `multiline` and give it a `width` to wrap at. `radius`, `offset`, `arrowSize`, `openDelay`, and `closeDelay` further tune the look and timing.

A longer tooltip that wraps onto several lines instead of stretching off the edge of the screen.

Long label

Source: Markdown

```
{% tooltip label='A longer tooltip that wraps onto several lines instead of stretching off
the edge of the screen.' multiline=true width=220 withArrow=true radius='md' %}Long label{%
endTooltip %}
```

Source: Python

```
component('aardvark', 'tooltip',
          label='A longer tooltip that wraps onto several lines instead of '
                'stretching off the edge of the screen.',
          multiline=True, width=220, withArrow=True, radius='md',
          children='Long label')
```

Force it open

`opened=true` keeps the tooltip showing without a hover — useful in docs or to call out a control:

Always visible

Pinned

Source: Markdown

```
{% tooltip label='Always visible' opened=true position='bottom' withArrow=true %}Pinned{%
endTooltip %}
```

Source: Python

```
component('aardvark', 'tooltip', label='Always visible',
          opened=True, position='bottom', withArrow=True, children='Pinned')
```

With other components

The target body can be any inline content — including another tag. Here a tooltip wraps a Badge so hovering the badge explains the status:

Deployed 4 minutes ago

[Live]

Source: Markdown

```
{% tooltip label='Deployed 4 minutes ago' position='top' withArrow=true %}{% badge
color='green' variant='light' %}Live{% endBadge %}{% endTooltip %}
```

Source: Python

```
component('aardvark', 'tooltip', label='Deployed 4 minutes ago',
          position='top', withArrow=True,
          children=component('aardvark', 'badge', color='green',
                             variant='light', children='Live'))
```

Attributes

Omit any attribute to take its Mantine default. The target is the block body (or, from Python, `children`).

Attribute	Valid values	Description
`label`	string	The floating text shown over the target.
`position`	`top` (default), `bottom`, `left`, `right`, plus the `-start` / `-end` variants	Edge the tooltip anchors to.

`color`	any theme color (`blue`, `dark`, `red`, `grape`, ...)	Background color of the tooltip.
`withArrow`	bare flag → `true`	Draw a small pointer at the anchored edge.
`arrowSize`	integer (pixels)	Arrow size.
`offset`	integer (pixels)	Gap between the target and the tooltip.
`multiline`	bare flag → `true`	Allow the label to wrap (pair with `width`).
`width`	integer (pixels)	Max width, for multiline labels.
`radius`	`xs`–`xl` or a number	Corner radius of the tooltip.
`openDelay`	integer (milliseconds)	Delay before showing on hover.
`closeDelay`	integer (milliseconds)	Delay before hiding after the pointer leaves.
`transition`	`fade`, `pop`, `scale`, `skew-up`, ...	Show/hide transition.
`transitionDuration`	integer (milliseconds)	Transition length.
`opened`	bare flag → `true`	Force the tooltip open — handy for documentation and screenshots.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Tooltip"]`, or through the Mantine Styles API classes. The tooltip mounts into a portal and its parts exist only while it is shown. The relevant classes:

```
/* Every rendered Tooltip carries this island marker */
[data-aardvark-island="Tooltip"] { }

/* Mantine Styles API classes */
.mantine-Tooltip-tooltip { }
.mantine-Tooltip-arrow { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert. (Unlike the other overlays, `attr` lands on the **target** here, not the floating tooltip — Mantine merges a Tooltip ref onto the target element, so the bubble can't be addressed directly.)

Saved 2 minutes ago

Hover me

Source: Markdown

```
{% tooltip label='Saved 2 minutes ago' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Hover me{% endTooltip %}
```

Source: Python

```
component('aardvark', 'tooltip', label='Saved 2 minutes ago', children='Hover me',
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Transition

`transition` is an **animation primitive**: it fades, slides, or scales its content in and out — keyed off a `mounted` flag. The animation runs when `mounted` changes, so it needs something to flip it. Like the modal and drawer, this tag ships with a **trigger button** that toggles `mounted` for you, so the enter/leave animation plays on click — no custom React required. `mounted` sets the initial state (default shown), `triggerLabel` sets the button's text, and the full set of Mantine presets is reachable from Markdown. Close the block with `{% endTransition %}`.

Use it as `{% transition %}…{% endTransition %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'transition', …)`.

Demonstrations

The block body is the animated content; pick a `transition` preset and a `duration`, then click the button to play it. `mounted=true` (the default) starts the content shown — click Toggle to slide it out, and again to slide it back in.

Preview — click Toggle to play it:

This block is wrapped in a Transition with the `slide-up` preset. Click the button above to slide it out of view and back in.

Source: Markdown

```
{% transition transition='slide-up' duration=300 mounted=true %}
Starts shown; click the trigger button to slide it out of view and back in.
{% endTransition %}
```

Source: Python

```
component('aardvark', 'transition',
          transition='slide-up', duration=300, mounted=True,
          children='Starts shown; click the trigger button to animate it.')
```

Presets

Any of Mantine's named presets work: `fade`, the `slide-*` family, `scale` / `scale-x` / `scale-y`, the `skew-*` and `rotate-*` families, and `pop`. Here the same block with `scale`, a longer `duration`, an explicit `timingFunction`, and a custom `triggerLabel` on the button.

Preview — click Scale it to play it:

The `scale` preset, played by the button above.

Source: Markdown

```
{% transition transition='scale' duration=400 timingFunction='ease-in-out' %}
…content…
{% endTransition %}
```

Source: Python

```
component('aardvark', 'transition',
          transition='scale', duration=400, timingFunction='ease-in-out',
          children='…content…')
```

Starting hidden

`mounted=false` starts the content hidden — the trigger button reveals it (and hides it again). Add `keepMounted=true` to keep the node in the DOM even while hidden, so its content stays present for SSR and crawlers.

Preview — starts hidden; click Reveal to fade it in:

Hidden until you click the button; kept in the DOM (so it's present for SSR and crawlers).

Source: Markdown

```
{% transition transition='fade' mounted=false keepMounted=true triggerLabel='Reveal' %}
Hidden until you click the button; kept in the DOM for SSR and crawlers.
{% endTransition %}
```

Source: Python

```
component('aardvark', 'transition',
          transition='fade', mounted=False, keepMounted=True, triggerLabel='Reveal',
          children='Hidden until you click the button; kept in the DOM for SSR and
crawlers.')
```

With other components

Transition is most natural wrapping a surface like Paper so the whole panel animates as a unit. The Preview at the top of this page already shows that pairing; here it is with a `pop` preset.

Preview — click Toggle to play it:

A Paper wrapped in a Transition with the `pop` preset.

Source: Markdown

```
{% transition transition='pop' duration=350 %}
{% paper withBorder=true p='md' radius='md' %}A Paper wrapped in a Transition.{% endPaper %}
{% endTransition %}
```

Source: Python

```
panel = component('aardvark', 'paper', withBorder=True, p='md', radius='md',
                  children='A Paper wrapped in a Transition.')
component('aardvark', 'transition', transition='pop', duration=350, children=panel)
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `keepMounted`) become `=True`.

Attribute	Valid values	Description
`transition`	`fade` (default) / `slide-up` / `slide-down` / `slide-left` / `slide-right` / `scale` / `scale-x` / `scale-y` / `skew-up` / `skew-down` / `rotate-left` / `rotate-right` / `pop` (and other Mantine presets)	The animation preset.
`duration`	An integer (ms, default `250`)	Animation length.
`mounted`	`true` (default) / `false`	The initial shown state; the trigger button toggles it. `mounted=false` starts hidden.
`triggerLabel`	Any string (default `Toggle`)	Text for the button that toggles the content.
`timingFunction`	A CSS easing value (e.g. `ease`, `ease-in-out`)	CSS easing for the animation.
`keepMounted`	`true` / `false` (default `false`)	Keep the node in the DOM even when hidden (so its content stays present for SSR / crawlers).
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Transition"]`. The animated content is positioned with inline styles, so it carries no Mantine part class — target the island mount or the content you place inside.

```
/* Every rendered Transition carries this island marker */
[data-aardvark-island="Transition"] { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Starts shown; click the trigger button to slide it out of view and back in.

Source: Markdown

```
{% transition transition='slide-up' duration=300 mounted=true attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Starts shown; click the trigger button to slide it out of view and back in.
{% endTransition %}
```

Source: Python

```
component('aardvark', 'transition',
          transition='slide-up', duration=300, mounted=True,
          children='Starts shown; click the trigger button to slide it out of view and back
in.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Data Display

Built-in tags for **showing** things — people, colors, statuses, keys, and content you want to reveal on demand. Each is a single Markdown tag with no setup; for the underlying component and every prop it accepts, follow the per-tag page.

- ArticleCard — blog-style article cards with a byline, avatar or initials fallback, date, badge, cover image, and five layout variants.
- Avatar — a user image with a graceful initials/placeholder fallback.
- Badge — labels, statuses, versions, and counts.
- BackgroundImage — a box with an image background and content overlaid on top.
- Card — flashy content cards with icons, images, variants, links, and a responsive grid.
- ColorSwatch — a square swatch that displays a single color.
- Icon — a Tabler, Font Awesome, image, or emoji glyph inline in text or a heading.
- Image — Mantine images with no-upscale sizing and a click-to-zoom lightbox.
- Indicator — a dot or count badge positioned over the corner of another element.
- Kbd — a keyboard key, for documenting shortcuts.
- Spoiler — collapse long content behind a show-more toggle.

Data & structure

- Accordion — collapsible sections whose bodies are full Markdown.
- Timeline — a vertical run of bullets joined by a line, each with a title and body.
- NumberFormatter — format a number with thousands separators, fixed decimals, and a prefix/suffix.
- RollingNumber — an animated counter that rolls to its value, re-animating each time the value changes.
- OverflowList — a single-line row of labels that collapses overflow into a “+N” counter, re-fitting on resize.

Accordion

A compound built-in tag for collapsible sections. The parent `{% accordion %}` wraps one or more `{% accordionSection title="…" %}` children; each section's `title` is the clickable control and everything between the section tags is its panel body. That body is ordinary **Markdown** — headings, lists, code, links, even nested `component(...)` calls — and renders as such. By default one panel is open at a time; add `multiple` to let several stay open.

Use it as `{% accordion %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'accordion', …)` — nesting `component('aardvark', 'accordionSection', …)` calls as its `children`.

Basic accordion

A single-open accordion with two sections. Each panel body is full Markdown.

Section One

Markdown content

This builds great!

Section Two

More Markdown content

- This bulleted list actually renders!
- See?

Source: Markdown

```
{% accordion %}
{% accordionSection title="Section One" %}
**Markdown content**

_This builds great!_
{% endAccordionSection %}
{% accordionSection title="Section Two" %}
**More Markdown content**

- This bulleted list actually renders!
- See?
{% endAccordionSection %}
{% endAccordion %}
```

Source: Python

```
component('aardvark', 'accordion', children=(
    component('aardvark', 'accordionSection', title='Section One',
```

```
                children='**Markdown content**\n\n_This builds great!_')
    + component('aardvark', 'accordionSection', title='Section Two',
                children='**More Markdown content**\n\n- This bulleted list actually
renders!\n- See?')
))
```

Multiple open panels

By default opening one panel closes the others. Add the bare `multiple` flag to let several stay open at once. In `multiple` mode, `defaultValue` takes a comma-separated list of section `value`s to open on load (a section's `value` defaults to a slug of its `title`, or set it explicitly).

Install

Run `npm install`.

Build

Run `vark build`.

Source: Markdown

```
{% accordion multiple defaultValue="install" %}
{% accordionSection title="Install" value="install" %}
Run `npm install`.
{% endAccordionSection %}
{% accordionSection title="Build" %}
Run `vark build`.
{% endAccordionSection %}
{% endAccordion %}
```

Source: Python

```
component('aardvark', 'accordion', multiple=True, defaultValue='install', children=(
    component('aardvark', 'accordionSection', title='Install', value='install',
              children='Run `npm install`.')
    + component('aardvark', 'accordionSection', title='Build',
                children='Run `vark build`.')
))
```

A single panel open on load

In single-open mode, `defaultValue` is one section `value` — the panel to expand on load. Here the second section is open from the start.

Overview

What the tool does, in a sentence.

Usage

How to run it, expanded by default.

Source: Markdown

```
{% accordion defaultValue="usage" %}
{% accordionSection title="Overview" value="overview" %}
What the tool does, in a sentence.
{% endAccordionSection %}
{% accordionSection title="Usage" value="usage" %}
How to run it, expanded by default.
{% endAccordionSection %}
{% endAccordion %}
```

Source: Python

```
component('aardvark', 'accordion', defaultValue='usage', children=(
    component('aardvark', 'accordionSection', title='Overview', value='overview',
              children='What the tool does, in a sentence.')
    + component('aardvark', 'accordionSection', title='Usage', value='usage',
                children='How to run it, expanded by default.')
))
```

With other components

A panel body is full Markdown, so it can hold any other built-in — here an icon in the title text and a nested card in the body.

What ships in the box

Everything you need to build and deploy:

Static output

Get started

Run `vark build` and you're live.

Source: Markdown

```
{% accordion %}
{% accordionSection title="What ships in the box" %}
Everything you need to build and deploy:

{% card title="Static output" subtitle="Plain HTML, CSS, and JS — host it anywhere."
icon="package" %}{% endCard %}
{% endAccordionSection %}
```

```
{% accordionSection title="Get started" %}
Run {% icon 'rocket' %} `vark build` and you're live.
{% endAccordionSection %}
{% endAccordion %}
```

Source: Python

```
box = ('Everything you need to build and deploy:\n\n'
       + component('aardvark', 'card', title='Static output',
                   subtitle='Plain HTML, CSS, and JS — host it anywhere.', icon='package'))
start = 'Run ' + component('aardvark', 'icon', 'rocket') + " `vark build` and you're live."
component('aardvark', 'accordion', children=(
    component('aardvark', 'accordionSection', title='What ships in the box', children=box)
    + component('aardvark', 'accordionSection', title='Get started', children=start)
))
```

Attributes

`{% accordion %}`

Attribute	Valid values	Description
`multiple`	bare flag (`multiple`)	Allow several panels open at once. Off by default (single-open).
`defaultValue`	a section `value`, or a comma-separated list with `multiple`	Panel(s) expanded on load. In single-open mode one `value`; in `multiple` mode a comma-separated list (`"a,b"`).

The look is fixed to Mantine's `contained` variant — there is no author-facing `variant`.

`{% accordionSection %}`

Attribute	Valid values	Description
`title`	string	The clickable control label.
`value`	string	Stable id for the panel, used by the parent's `defaultValue`. Defaults to a slug of `title` (deduplicated if two titles collide).

The body between `{% accordionSection %}` and `{% endAccordionSection %}` is the panel content (full Markdown). In the Python form it is the `children` argument.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Accordion"]` — or through the Mantine Styles API classes (`.mantine-Accordion-root` and its inner parts):

```
/* Every rendered Accordion carries this island marker */
[data-aardvark-island="Accordion"] { }

/* Mantine Styles API class on the root element */
.mantine-Accordion-root { }
.mantine-Accordion-item { }
.mantine-Accordion-control { }
.mantine-Accordion-panel { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered accordion root. (Scalar attributes beyond the ones it handles — like `multiple` to keep several panels open, or a `defaultValue` — pass straight through to the underlying Mantine `Accordion`; `attr={…}` is the channel for raw HTML attributes.)

Overview

A single panel of body text, open by default.

Details

A second panel that can stay open alongside it.

Source: Markdown

```
{% accordion multiple defaultValue="overview" attr={'data-analytics': 'faq', 'aria-label':
'Frequently asked questions'} %}
{% accordionSection title="Overview" %}
A single panel of body text, open by default.
{% endAccordionSection %}
{% accordionSection title="Details" %}
A second panel that can stay open alongside it.
{% endAccordionSection %}
{% endAccordion %}
```

Source: Python

```
component('aardvark', 'accordion', multiple=True, defaultValue='overview',
          attr={'data-analytics': 'faq', 'aria-label': 'Frequently asked questions'},
children=(
    component('aardvark', 'accordionSection', title='Overview', value='overview',
              children='A single panel of body text, open by default.')
    + component('aardvark', 'accordionSection', title='Details',
                children='A second panel that can stay open alongside it.')
```

```
))
```

AreaChart

An **area chart** — like a line chart with the region under each line filled, good for showing volume or stacked totals. Give it the rows in `data` and the areas in `series` (`{name, color}`), with `dataKey` for the x-axis. Renders in the browser and hydrates into an interactive island.

Use it as `{% areachart %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'areachart', …)`.

A basic area chart

Source: Markdown

```
{% areachart h=260 dataKey='month' curveType='monotone' withLegend=true data='[
  {"month":"Jan","Requests":1200,"Errors":40},
  {"month":"Feb","Requests":1800,"Errors":30},
  {"month":"Mar","Requests":1600,"Errors":55},
  {"month":"Apr","Requests":2400,"Errors":25}
]' series='[{"name":"Requests","color":"blue.6"},{"name":"Errors","color":"red.6"}]' %}
```

Source: Python

```
component('aardvark', 'areachart', h=260, dataKey='month', curveType='monotone',
withLegend=True,
          data='[{"month":"Jan","Requests":1200,"Errors":40}, …]',
          series='[{"name":"Requests","color":"blue.6"},{"name":"Errors","color":"red.6"}]')
```

Attributes

Attribute	Valid values	Description
`data`	JSON array of row objects	The chart rows.
`series`	JSON array of `{name, color}`	One area per entry.
`dataKey`	string	The row field used for the x-axis.
`h`	integer (px, default `300`)	Chart height.
`curveType`	`monotone`, `linear`, `natural`, `step`, …	Curve interpolation.
`unit`	string	Unit appended to axis/tooltip values.
`withLegend`	bool (`true` / `false`)	Show the legend.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="AreaChart"]` (or the more specific `[data-aardvark-lib="charts"][data-aardvark-island="AreaChart"]` when several libraries share the page) — or through the Mantine Styles API classes (`.mantine-AreaChart-root` and its inner parts):

```
/* Every rendered AreaChart carries this island marker */
[data-aardvark-island="AreaChart"] { }

/* Scope by library + island when you have several libraries in play */
[data-aardvark-lib="charts"][data-aardvark-island="AreaChart"] { }

/* Mantine Styles API class on the root element */
.mantine-AreaChart-root { }
.mantine-AreaChart-container { }
.mantine-AreaChart-axis { }
.mantine-AreaChart-grid { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% areachart h=260 dataKey='month' withLegend=true data='[
  {"month":"Jan","Requests":1200,"Errors":40},
  {"month":"Feb","Requests":1800,"Errors":30}
]' series='[{"name":"Requests","color":"blue.6"},{"name":"Errors","color":"red.6"}]'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'areachart', h=260, dataKey='month', withLegend=True,
          data='[{"month":"Jan","Requests":1200,"Errors":40},{"month":"Feb","Requests":1800,"Errors":3
0}]',
          series='[{"name":"Requests","color":"blue.6"},{"name":"Errors","color":"red.6"}]',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

ArticleCard

A **built-in** tag for blog-style article cards — the card the `{% taxonomy %}` article listing renders one-per-post, available standalone for hand-built index pages. You give it a flat attribute set — title, teaser `description`, a **byline** with an avatar, a date, an optional badge, tags, and an optional cover image — and pick one of five layouts: a horizontal media card, a footer-metadata card, a full-bleed background-image card, a vertical card, and a plain grid card. Wrap a set of them in `{% cardGrid %}` for a responsive grid, or call it from Python via `component('aardvark', 'articleCard', …)`. (The variants match the mantine.dev article-cards gallery if you want a visual reference.)

Cards **degrade gracefully**: omit `authorAvatar` and the byline shows the author's **initials** in a colored disc; omit the author entirely and the byline row is dropped; omit `image` and image-led variants render as clean text cards rather than leaving a hole.

Demonstrations

Horizontal

`variant="horizontal"` (the default) is the gallery's classic article card: cover image on top, the badge floating over its corner in a gradient chip, title, a four-line teaser, and a footer with the byline on the left and the date on the right:

Introducing Ask AI, the reader assistant

A native Ask AI panel that answers reader questions from your own docs, with cited sources and metered, dollar-based billing.

Source: Markdown

```
{% articleCard variant="horizontal" title="Introducing Ask AI, the reader assistant"
href="/blog/introducing-ask-ai/" image="/landscape.jpg" imageAlt="" badge="Product"
authorName="The Aardvark team" authorAvatar="/favicon.svg" date="2026-05-05" tags="ai,
product" %}
A native Ask AI panel that answers reader questions from your own docs, with cited
sources and metered, dollar-based billing.
{% endArticleCard %}
```

Source: Python

```
component('aardvark', 'articleCard', variant='horizontal',
          title='Introducing Ask AI, the reader assistant',
href='/blog/introducing-ask-ai/',
          image='/landscape.jpg', imageAlt='', badge='Product',
          authorName='The Aardvark team', authorAvatar='/favicon.svg',
          date='2026-05-05', tags='ai, product',
          children='A native Ask AI panel that answers reader questions from your own '
```

```
                    'docs, with cited sources and metered, dollar-based billing.')
```

Footer

`variant="footer"` leads with an outline badge and the byline block, and moves the teaser into a separated border-top footer strip — good for uniform grids. Note the **initials fallback**: no `authorAvatar` here, so the byline draws a disc from the author's initials:

Why Aardvark rebuilds every page (for now)

Where the dev loop's speed actually comes from, and why per-page skipping is a harder promise than it looks.

Source: Markdown

```
{% articleCard variant="footer" title="Why Aardvark rebuilds every page (for now)"
href="/blog/how-incremental-builds-work/" badge="Engineering" authorName="Aardvark
engineering" date="2026-06-02" tags="build, performance" %}
Where the dev loop's speed actually comes from, and why per-page skipping is a harder
promise than it looks.
{% endArticleCard %}
```

Background

`variant="background"` is the tall photo card: the badge renders as an uppercase category caption over a large white title, with an optional `cta` button pinned at the bottom — the hero treatment. With no `image`, it falls back to a plain text card:

Multi-language docs in practice

Source: Markdown

```
{% articleCard variant="background" title="Multi-language docs in practice"
href="/blog/multi-language-docs-in-practice/" image="/landscape.jpg" imageAlt=""
badge="How-to" cta="Read article" %}{% endArticleCard %}
```

Vertical

`variant="vertical"` is the side-by-side row — image on the left (stacking on small screens), an uppercase category caption, the title, and a compact `author • date` line. Here two of them ride a `{% cardGrid %}`:

Anatomy of a Mantine island

vark 0.9 release roundup

Source: Markdown

```
{% cardGrid colsBase=1 colsSm=2 %}
{% articleCard variant="vertical" title="Anatomy of a Mantine island"
href="/blog/anatomy-of-a-mantine-island/" image="/img/sample-landscape.svg" imageAlt=""
badge="Deep dive" authorName="Aardvark engineering" authorAvatar="/img/sample-square.svg"
date="2026-06-18" tags="components, engineering" %}{% endArticleCard %}
{% articleCard variant="vertical" title="vark 0.9 release roundup"
href="/blog/vark-0-9-release-roundup/" image="/landscape.jpg" imageAlt="" badge="Release"
authorName="The Aardvark team" authorAvatar="/favicon.svg" date="2026-05-30" tags="release,
cli" %}{% endArticleCard %}
{% endCardGrid %}
```

Plain

`variant="plain"` is the grid tile: a rounded 16:9 cover (when given), a small uppercase date, and the title, with the border materializing on hover. It is deliberately minimal — no teaser text, even if a `description` is supplied:

Designing the honest AI meter

Source: Markdown

```
{% articleCard variant="plain" title="Designing the honest AI meter"
href="/blog/designing-the-honest-ai-meter/" date="2026-07-08" tags="pricing, product" %}{%
endArticleCard %}
```

Attributes

Set what you need, omit the rest.

Attribute	Valid values	Description
`title`	string	The article headline. Also the accessible link name when `href` is set.
`href`	URL	Make the whole card a link to the article.
`url`	URL	Alias for `href`, used when `href` is absent (handy when reusing a data source that names the field `url`).

`description`	string	The teaser text — the four-line body of `horizontal`, the footer strip of `footer`. `vertical` and `plain` are deliberately compact and omit it; `background` omits it on the photo card but shows it in the no-image text fallback. The tag **body** does the same job — use whichever reads better; with both given, the attribute wins and the body is ignored.
`image`	image URL	Cover image — on top (`horizontal`/`footer`), on the left (`vertical`), a rounded 16:9 tile (`plain`), or the full background (`background`). Omitted, the card renders as a text card.
`imageAlt`	string	Alt text for the image (use `imageAlt=""` for a decorative image).
`badge`	string	A badge label — a gradient chip over the cover (`horizontal`), an outline chip (`footer`), or an uppercase category caption (`background`/`vertical`).
`badgeColor`	any theme/CSS color	Badge color (chip-style variants; `horizontal` switches from the gradient to a filled chip).
`authorName`	string	The byline. Omitted, the byline row is dropped.
`authorAvatar`	image URL	The byline avatar. Omitted, the byline falls back to the author's **initials** in a colored disc.
`date`	`YYYY-MM-DD`, optionally with a time	The article date shown on the card.
`dateDisplay`	string	A pre-formatted date string to show instead of the default formatting of `date`.
`variant`	`horizontal`, `footer`, `background`, `vertical`, `plain`	Card layout.
`cta`	string	Call-to-action label — rendered by the `background` variant only (as a button-styled affordance inside the card's single link).

`tags`	comma-separated string	Accepted for payload parity with cards supplied through a taxonomy article listing (whose islands filter on the data) — a **standalone** card neither renders nor filters by them.
`onclick`	JS expression	A JS click handler on the card root (a raw HTML attribute; in Python it rides `attr={'onclick': …}`).
`attr={…}`	mapping	Forward raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks, event handlers — onto the rendered card root. See Injecting Attributes below.

CSS Selectors

Each card is a `[data-aardvark-articlecard]` root carrying its layout in `data-variant`; inner parts use the `.aardvark-articlecard-*` prefix:

```
[data-aardvark-articlecard]              /* one article card */
[data-aardvark-articlecard][data-variant="footer"]  /* one layout variant */
.aardvark-articlecard-title              /* the (linked) title */
.aardvark-articlecard-authorrow          /* byline: avatar + name + date */
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks — onto the rendered card root:

vark 0.9 release roundup

A self-generating CLI reference, a faster dev loop, and the best of the 0.8 line.

Source: Markdown

```
{% articleCard variant="plain" title="vark 0.9 release roundup"
href="/blog/vark-0-9-release-roundup/" date="2026-05-30" attr={'data-analytics':
'blog-card', 'aria-label': 'Release roundup'} %}
A self-generating CLI reference, a faster dev loop, and the best of the 0.8 line.
{% endArticleCard %}
```

Source: Python

```
component('aardvark', 'articleCard', variant='plain', title='vark 0.9 release roundup',
          href='/blog/vark-0-9-release-roundup/', date='2026-05-30',
          children='A self-generating CLI reference, a faster dev loop, and the '
                   'best of the 0.8 line.',
          attr={'data-analytics': 'blog-card', 'aria-label': 'Release roundup'})
```

Avatar

A user avatar: an image that degrades gracefully to initials or a placeholder when the image is missing or fails to load. Point `src` at an image; when there's no `src` (or it can't load), the block body — or, with a `name`, auto-generated initials — is shown instead. It ships with Aardvark, so it's a single tag with no setup.

Use it as `{% avatar %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'avatar', …)`.

Demonstrations

Image, initials, and a body fallback

Three ways to fill an avatar: a real image (`src` + `alt`), auto-generated initials from a `name`, or an explicit body. With `color='initials'`, the color is derived deterministically from the `name`.

AL

Source: Markdown

```
{% avatar src='/avatar.png' alt='Ada Lovelace' %}
{% avatar name='Ada Lovelace' %}
{% avatar name='Grace Hopper' color='initials' %}
{% avatar %}AL{% endAvatar %}
```

Source: Python

```
component('aardvark', 'avatar', src='/avatar.png', alt='Ada Lovelace')
component('aardvark', 'avatar', name='Ada Lovelace')
component('aardvark', 'avatar', name='Grace Hopper', color='initials')
component('aardvark', 'avatar', children='AL')
```

Sizes and radius

`size` takes `xs`–`xl` or any CSS value; `radius` takes `xs`–`xl`, or `'100%'` (the default) for a circle. A smaller `radius` squares off the corners.

Source: Markdown

```
{% avatar name='XS' size='xs' %}
{% avatar name='MD' size='md' %}
{% avatar name='XL' size='xl' %}
{% avatar name='SQ' radius='sm' %}
```

Source: Python

```
component('aardvark', 'avatar', name='XS', size='xs')
component('aardvark', 'avatar', name='MD', size='md')
component('aardvark', 'avatar', name='XL', size='xl')
component('aardvark', 'avatar', name='SQ', radius='sm')
```

Initials and color

With no `src`, `name` drives the initials. Pass any theme color to `color`, or use `color='initials'` to derive a stable color from the name.

Source: Markdown

```
{% avatar name='Ada Lovelace' color='initials' %}
{% avatar name='Grace Hopper' color='initials' %}
{% avatar name='Alan Turing' color='grape' %}
```

Source: Python

```
component('aardvark', 'avatar', name='Ada Lovelace', color='initials')
component('aardvark', 'avatar', name='Grace Hopper', color='initials')
component('aardvark', 'avatar', name='Alan Turing', color='grape')
```

Variants and gradient

`variant` accepts `filled`, `light`, `outline`, `transparent`, `white`, `default`, and `gradient`. With `variant='gradient'`, set any of `gradientFrom` / `gradientTo` / `gradientDeg` to control the gradient. `autoContrast` picks a readable text color for the background.

Source: Markdown

```
{% avatar name='F' variant='filled' color='blue' %}
{% avatar name='L' variant='light' color='blue' %}
{% avatar name='O' variant='outline' color='blue' %}
{% avatar name='W' variant='white' color='blue' %}
{% avatar name='G' variant='gradient' gradientFrom='indigo' gradientTo='cyan' gradientDeg=90
%}
```

Source: Python

```
component('aardvark', 'avatar', name='F', variant='filled', color='blue')
component('aardvark', 'avatar', name='L', variant='light', color='blue')
component('aardvark', 'avatar', name='O', variant='outline', color='blue')
component('aardvark', 'avatar', name='W', variant='white', color='blue')
component('aardvark', 'avatar', name='G', variant='gradient',
          gradientFrom='indigo', gradientTo='cyan', gradientDeg=90)
```

With other components

An avatar is a natural target for an `{% indicator %}`, which puts a count or status dot over its corner:

3

Source: Markdown

```
{% indicator label='3' color='red' %}
{% avatar name='Ada Lovelace' %}
{% endIndicator %}

{% indicator color='green' processing=true %}
{% avatar name='Grace Hopper' %}
{% endIndicator %}
```

Source: Python

```
component('aardvark', 'indicator', label='3', color='red',
          children=component('aardvark', 'avatar', name='Ada Lovelace'))
component('aardvark', 'indicator', color='green', processing=True,
          children=component('aardvark', 'avatar', name='Grace Hopper'))
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`src`	Image URL	The avatar image. When missing or broken, the fallback (body or initials) shows.
`name`	string	The user's name — used for initials and, with `color='initials'`, a deterministic color.
`alt`	string	Image `alt` text (and the placeholder's title). Always set this for accessibility.
`size`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	The avatar's width and height.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value (`'100%'` is the default circle)	Corner rounding.

`color`	any theme color, or `initials`	Background/text color. `initials` derives a stable color from `name`.
`variant`	`filled`, `light`, `outline`, `transparent`, `white`, `default`, `gradient`	Visual style.
`autoContrast`	bool flag (default `false`)	Auto-pick a readable text color for the background.
`gradientFrom`	any theme color	Gradient start color (with `variant='gradient'`).
`gradientTo`	any theme color	Gradient end color (with `variant='gradient'`).
`gradientDeg`	integer	Gradient angle in degrees (with `variant='gradient'`).
body	text	Explicit fallback shown when there is no `src` (or it fails to load).

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Avatar"]` — or through the Mantine Styles API classes (`.mantine-Avatar-root` and its inner parts):

```
/* Every rendered Avatar carries this island marker */
[data-aardvark-island="Avatar"] { }

/* Mantine Styles API class on the root element */
.mantine-Avatar-root { }
.mantine-Avatar-placeholder { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% avatar name='Ada Lovelace' attr={'onclick': '''
```

```
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'avatar', name='Ada Lovelace', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

BackgroundImage

A box that displays an image as its **background**, with your content laid over it. Point `src` at the image, round the corners with `radius`, and put whatever you like in the block body — it renders on top of the background. The Mantine spacing and sizing system is forwarded, so you can pad and size the box without writing CSS.

Use it as `{% backgroundimage %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'backgroundimage', …)`. Close the tag with `{% endBackgroundimage %}` (lowercase `image`).

Demonstrations

Overlaid content

The block body is the overlaid content. Here a white heading sits over the image; `p='xl'` pads it away from the edges and `mah='180'` caps the height.

Overlaid heading

Source: Markdown

```
{% backgroundimage src='/cover.jpg' radius='md' p='xl' mah='180' %}
{% text fw='700' c='white' size='xl' %}Overlaid heading{% endText %}
{% endBackgroundimage %}
```

Source: Python

```
component('aardvark', 'backgroundimage',
          src='/cover.jpg', radius='md', p='xl', mah='180',
          children=component('aardvark', 'text', fw='700', c='white', size='xl',
                             children='Overlaid heading'))
```

Radius and sizing

`radius` rounds the corners (`xs`–`xl` or any CSS value); the spacing/sizing props (`w`, `h`, `maw`, `mah`, `p`, `m`, …) set and pad the box. A fully round `radius` plus a fixed `w`/`h` makes a circular cover.

Source: Markdown

```
{% backgroundimage src='/cover.jpg' radius='100%' w='120' h='120' %}
{% endBackgroundimage %}
```

Source: Python

```
component('aardvark', 'backgroundimage',
          src='/cover.jpg', radius='100%', w='120', h='120')
```

With other components

Overlay any built-in tag. Here a `{% text %}` heading and a caption stack on the background:

Source: Markdown

```
{% backgroundimage src='/cover.jpg' radius='md' p='xl' mah='200' %}
{% text fw='700' c='white' size='xl' %}Mountain retreat{% endText %}
{% text c='white' size='sm' %}A box with content overlaid on an image.{% endText %}
{% endBackgroundimage %}
```

Source: Python

```
heading = component('aardvark', 'text', fw='700', c='white', size='xl',
                    children='Mountain retreat')
caption = component('aardvark', 'text', c='white', size='sm',
                    children='A box with content overlaid on an image.')
component('aardvark', 'backgroundimage',
          src='/cover.jpg', radius='md', p='xl', mah='200',
          children=heading + caption)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`src`	Image URL (**required**)	The background image.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	Corner rounding.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	Mantine size token (`xs`–`xl`) or any CSS value	Outer margin (all / top / bottom / left / right / horizontal / vertical).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	Mantine size token (`xs`–`xl`) or any CSS value	Inner padding (all / top / bottom / left / right / horizontal / vertical).
`w`, `h`	any CSS value	Box width and height.

`miw`, `mih`	any CSS value	Minimum width and height.
`maw`, `mah`	any CSS value	Maximum width and height.
body	Markdown / components	The content overlaid on top of the background.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="BackgroundImage"]` — or through the Mantine Styles API classes (`.mantine-BackgroundImage-root` and its inner parts):

```
/* Every rendered BackgroundImage carries this island marker */
[data-aardvark-island="BackgroundImage"] { }

/* Mantine Styles API class on the root element */
.mantine-BackgroundImage-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% backgroundimage src='https://images.unsplash.com/photo-1507525428034-b723cf961d3e?w=800'
radius='md' p='xl' mah='180' attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'backgroundimage',
          src='https://images.unsplash.com/photo-1507525428034-b723cf961d3e?w=800',
          radius='md', p='xl', mah='180', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''})
```

Badge

A **built-in** tag for small labels — statuses, versions, counts, and tags. It ships with Aardvark, so a badge is a single tag with no setup. The label is the block body or a `text` param. Use it as `{% badge %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'badge', …)`.

Demonstrations

Variants

Every Mantine badge variant — `filled` (the default), `light`, `outline`, `dot`, `transparent`, `white`, and `default`:

[filled] [light] [outline] [dot] [transparent] [white] [default]

Source: Markdown

```
{% badge %}filled{% endBadge %}
{% badge variant='light' %}light{% endBadge %}
{% badge variant='outline' %}outline{% endBadge %}
{% badge variant='dot' %}dot{% endBadge %}
{% badge variant='transparent' %}transparent{% endBadge %}
{% badge variant='white' %}white{% endBadge %}
{% badge variant='default' %}default{% endBadge %}
```

Source: Python

```
for v in ('filled', 'light', 'outline', 'dot', 'transparent', 'white', 'default'):
    page.print(component('aardvark', 'badge', variant=v, children=v))
```

Colors

`color` takes any theme color (`blue`, `green`, `grape`, …) or any CSS color:

[green] [grape] [orange] [gray]

Source: Markdown

```
{% badge color='green' %}green{% endBadge %}
{% badge color='grape' %}grape{% endBadge %}
{% badge color='orange' %}orange{% endBadge %}
{% badge color='gray' %}gray{% endBadge %}
```

Source: Python

```
for c in ('green', 'grape', 'orange', 'gray'):
    page.print(component('aardvark', 'badge', color=c, children=c))
```

Sizes and radius

`size` and `radius` both take `xs`–`xl`:

[xs] [sm] [md] [lg] [xl]

Source: Markdown

```
{% badge size='xs' %}xs{% endBadge %}
{% badge size='sm' %}sm{% endBadge %}
{% badge size='md' %}md{% endBadge %}
{% badge size='lg' %}lg{% endBadge %}
{% badge size='xl' radius='xl' %}xl{% endBadge %}
```

Source: Python

```
for s in ('xs', 'sm', 'md', 'lg', 'xl'):
    page.print(component('aardvark', 'badge', size=s, children=s))
```

Gradient

`variant='gradient'` paints the badge with a gradient; set any of `gradientFrom`, `gradientTo`, and `gradientDeg` to control the endpoints and angle:

[Pro]

Source: Markdown

```
{% badge text='Pro' variant='gradient' gradientFrom='indigo' gradientTo='cyan'
gradientDeg=90 %}
```

Source: Python

```
component('aardvark', 'badge', text='Pro', variant='gradient',
          gradientFrom='indigo', gradientTo='cyan', gradientDeg=90)
```

Circle, sections, and full width

`circle` renders a round badge — good for a single character or count. `leftSection`/`rightSection` take text shown before/after the label, and `fullWidth` stretches the badge to its container:

[9] [Updates] [2.0] [full width]

Source: Markdown

```
{% badge color='red' circle=true size='lg' %}9{% endBadge %}
{% badge variant='light' color='blue' rightSection='12' %}Updates{% endBadge %}
{% badge variant='light' color='teal' leftSection='v' %}2.0{% endBadge %}
{% badge fullWidth=true color='grape' %}full width{% endBadge %}
```

Source: Python

```
component('aardvark', 'badge', color='red', circle=True, size='lg', children='9')
component('aardvark', 'badge', variant='light', color='blue', rightSection='12',
```

```
children='Updates')
component('aardvark', 'badge', variant='light', color='teal', leftSection='v',
children='2.0')
component('aardvark', 'badge', fullWidth=True, color='grape', children='full width')
```

With other components

A badge sits naturally inside a heading, list, or another component — here it labels a {% card %} title and rides alongside an {% icon %}:

Status: [Stable]

Releases

The latest build [New] ships today.

Source: Markdown

```
Status: {% icon "circle-check" color="green" %} {% badge color='green' variant='light'
%}Stable{% endBadge %}

{% card title="Releases" %}
The latest build {% badge color='green' %}New{% endBadge %} ships today.
{% endCard %}
```

Source: Python

```
body = "The latest build " + component('aardvark', 'badge', color='green', children='New') +
" ships today."
page.print(component('aardvark', 'card', title='Releases', children=body))
```

Attributes

Attribute	Valid values	Description
(body) / `text`	string	The label. Use the block body, or the `text` param when not using a body.
`variant`	`filled` (default), `light`, `outline`, `dot`, `transparent`, `white`, `default`, `gradient`	Visual style.
`color`	any theme color (`blue`, `green`, `grape`, ...) or any CSS color	Badge color.
`size`	`xs`, `sm`, `md`, `lg`, `xl`	Badge size.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`	Corner rounding.
`fullWidth`	bool flag (`true`/`false`)	Stretch to the container's width. Defaults to `false`.
`circle`	bool flag (`true`/`false`)	Render as a circle — good for a single character or count. Defaults to `false`.
`autoContrast`	bool flag (`true`/`false`)	Auto-pick a readable label color for the background. Defaults to `false`.
`leftSection`	string	Text shown before the label.
`rightSection`	string	Text shown after the label.
`gradientFrom`	any theme/CSS color	Gradient start color, used with `variant='gradient'`.
`gradientTo`	any theme/CSS color	Gradient end color, used with `variant='gradient'`.
`gradientDeg`	integer (degrees)	Gradient angle, used with `variant='gradient'`.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Badge"]` — or through the Mantine Styles API classes (`.mantine-Badge-root` and its inner parts):

```
/* Every rendered Badge carries this island marker */
[data-aardvark-island="Badge"] { }

/* Mantine Styles API class on the root element */
.mantine-Badge-root { }
.mantine-Badge-label { }
.mantine-Badge-section { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

[Stable]

Source: Markdown

```
{% badge color='green' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Stable{% endBadge %}
```

Source: Python

```
component('aardvark', 'badge', color='green', children='Stable', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

BarChart

A **bar chart** for comparing values across categories. Give it the rows in `data` (a JSON array of objects) and the bars in `series` (a JSON array of `{name, color}`), with `dataKey` naming the category axis. Renders in the browser and hydrates into an interactive island.

Use it as `{% barchart %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'barchart', …)`.

A basic bar chart

Source: Markdown

```
{% barchart h=260 dataKey='product' withLegend=true data='[
  {"product":"Free","Q1":40,"Q2":55},
  {"product":"Pro","Q1":80,"Q2":120},
  {"product":"Team","Q1":30,"Q2":48}
]' series='[{"name":"Q1","color":"violet.6"},{"name":"Q2","color":"blue.6"}]' %}
```

Source: Python

```
component('aardvark', 'barchart', h=260, dataKey='product', withLegend=True,
          data='[{"product":"Free","Q1":40,"Q2":55}, …]',
          series='[{"name":"Q1","color":"violet.6"},{"name":"Q2","color":"blue.6"}]')
```

Attributes

Attribute	Valid values	Description
`data`	JSON array of row objects	The chart rows.
`series`	JSON array of `{name, color}`	One bar series per entry.
`dataKey`	string	The row field used for the category axis.
`h`	integer (px, default `300`)	Chart height.
`unit`	string	Unit appended to axis/tooltip values.
`withLegend`	bool (`true` / `false`)	Show the legend.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="BarChart"]` (or the more specific `[data-aardvark-lib="charts"][data-aardvark-island="BarChart"]` when several libraries share the page) — or through the Mantine Styles API classes (`.mantine-BarChart-root` and its inner parts):

```
/* Every rendered BarChart carries this island marker */
[data-aardvark-island="BarChart"] { }

/* Scope by library + island when you have several libraries in play */
[data-aardvark-lib="charts"][data-aardvark-island="BarChart"] { }

/* Mantine Styles API class on the root element */
.mantine-BarChart-root { }
.mantine-BarChart-container { }
.mantine-BarChart-axis { }
.mantine-BarChart-bar { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% barchart h=260 dataKey='product' withLegend=true data='[
  {"product":"Free","Q1":40,"Q2":55},
  {"product":"Pro","Q1":80,"Q2":120}
]' series='[{"name":"Q1","color":"violet.6"},{"name":"Q2","color":"blue.6"}]'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'barchart', h=260, dataKey='product', withLegend=True,
          data='[{"product":"Free","Q1":40,"Q2":55},{"product":"Pro","Q1":80,"Q2":120}]',
          series='[{"name":"Q1","color":"violet.6"},{"name":"Q2","color":"blue.6"}]',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Card

A **built-in** tag built from Mantine's `Card` (plus `Image`, `BackgroundImage`, `Badge`, and `ThemeIcon`) behind one tag pair. One card does pretty much anything a card can: a title and a **Markdown** body, an icon, a cover or full-bleed background image, badges, an accent color, gradient/glass/stat looks, and a whole-card outbound link. Wrap a set of them in `{% cardGrid %}` for a responsive grid. Use it as `{% card %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'card', …)` (and `component('aardvark', 'cardGrid', …)`).

Cards are **flashy by default** — hover lift, cover-image zoom, an animated sheen on link cards, accent borders — and all of it is reduced-motion-safe and looks right in light and dark.

Demonstrations

Basic card

A title and a Markdown body. Standalone, a card is a full-width block:

What is Aardvark?

A documentation-focused static site generator — **Markdown in, static HTML out**, with interactive Mantine islands.

Source: Markdown

```
{% card title="What is Aardvark?" %}
A documentation-focused static site generator — **Markdown in, static HTML out**, with
interactive [Mantine](https://mantine.dev) islands.
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', title='What is Aardvark?',
          children='A documentation-focused static site generator — **Markdown in, '
                   'static HTML out**, with interactive [Mantine](https://mantine.dev)
islands.')
```

Icons

`icon="…"` takes any **Tabler icon** name (kebab-case, e.g. `rocket`, `brand-github`, `bolt`). Tabler icons are **lazy-loaded** — only the icons a page actually uses are fetched from a CDN at view time, so the full ~5,800-icon library is available without bloating the page. `icon=` also accepts a **Font Awesome** class (`icon="fa-solid fa-rocket"`, `icon="fab fa-github"` — needs `theme.fontawesome: true`), an emoji, an image URL, or inline `<svg>`. `accent="…"` colors the icon chip, accent bar, hover border, and CTA.

Fast builds

Incremental Markdown rendering and one esbuild pass.

Secure by default

Author-trusted, but hardened — escaped props, sanitized links.

Open source

Browse the icon set at tabler.io/icons.

Source: Markdown

```
{% cardGrid %}
{% card title="Fast builds" icon="bolt" accent="yellow" %}
Incremental Markdown rendering and one esbuild pass.
{% endCard %}
{% card title="Secure by default" icon="shield" accent="teal" %}
Author-trusted, but hardened — escaped props, sanitized links.
{% endCard %}
{% card title="Open source" icon="brand-github" accent="grape" %}
Browse the icon set at [tabler.io/icons](https://tabler.io/icons).
{% endCard %}
{% endCardGrid %}
```

Source: Python

```
cards = component('aardvark', 'card', title='Fast builds', icon='bolt', accent='yellow',
                  children='Incremental Markdown rendering and one esbuild pass.')
cards += component('aardvark', 'card', title='Secure by default', icon='shield',
accent='teal',
                   children='Author-trusted, but hardened — escaped props, sanitized
links.')
cards += component('aardvark', 'card', title='Open source', icon='brand-github',
```

```
accent='grape',
                    children='Browse the icon set at
[tabler.io/icons](https://tabler.io/icons).')
page.print(component('aardvark', 'cardGrid', children=cards))
```

Link cards

Add `href` and the **whole card** becomes a link (with a hover lift and an animated sheen). External URLs open in a new tab and get an external-link icon automatically; internal links just navigate. A `cta` adds a **button** at the bottom (use `ctaVariant="link"` for a plain text link with an arrow instead). `badge`/`badges` work on any card.

🚀Quickstart

[Popular]

Build your first site in a minute.

❖Components

Every Mantine component, callable from Markdown.

Mantine

The React component library under the hood.

Source: Markdown

```
{% cardGrid %}
{% card title="Quickstart" icon="rocket" badge="Popular" badgeColor="grape"
href="/docs/#quickstart" cta="Get going" %}
Build your first site in a minute.
{% endCard %}
{% card title="Components" icon="components" href="/components/" cta="Browse" %}
Every Mantine component, callable from Markdown.
{% endCard %}
{% card title="Mantine" icon="fab fa-react" href="https://mantine.dev" cta="Visit" %}
The React component library under the hood.
{% endCard %}
{% endCardGrid %}
```

Source: Python

```
cards = component('aardvark', 'card', title='Quickstart', icon='rocket', badge='Popular',
                  badgeColor='grape', href='/docs/#quickstart', cta='Get going',
                  children='Build your first site in a minute.')
cards += component('aardvark', 'card', title='Components', icon='components',
                   href='/components/', cta='Browse',
                   children='Every Mantine component, callable from Markdown.')
page.print(component('aardvark', 'cardGrid', children=cards))
```

Cover images

`image="…"` puts a full-bleed cover image at the top (it zooms on hover for link cards). Pair it with `badge` for a label, and `alt` for accessibility (`alt=""` for a purely decorative image):

Theming

[Guide]

Colors, fonts, and layout — make it yours.

Image

[New]

Zoom into any figure with a click-to-open lightbox.

Source: Markdown

```
{% cardGrid colsBase=1 colsSm=2 %}
{% card title="Theming" image="/img/sample-landscape.svg" alt="" badge="Guide"
href="/components/" %}
Colors, fonts, and layout — make it yours.
{% endCard %}
{% card title="Image" image="/landscape.jpg" alt="" badge="New" badgeColor="grape"
href="/components/data-display/image/" %}
Zoom into any figure with a click-to-open lightbox.
{% endCard %}
{% endCardGrid %}
```

Source: Python

```
cards = component('aardvark', 'card', title='Theming', image='/img/sample-landscape.svg',
                  alt='', badge='Guide', href='/components/',
                  children='Colors, fonts, and layout — make it yours.')
cards += component('aardvark', 'card', title='Image', image='/landscape.jpg', alt='',
```

```
                    badge='New', badgeColor='grape', href='/components/data-display/image/',
                    children='Zoom into any figure with a click-to-open lightbox.')
page.print(component('aardvark', 'cardGrid', colsBase=1, colsSm=2, children=cards))
```

Background-image cards

`variant="image"` makes the image the **full card background**, with the text laid over it. A gradient scrim keeps the title and body readable on any photo (WCAG AA), in both color schemes:

Deploy anywhere

Build to static HTML and host it anywhere — a full-bleed hero with a readable scrim.

Source: Markdown

```
{% card variant="image" image="/landscape.jpg" alt="" title="Deploy anywhere" cta="Ship it"
href="/docs/#quickstart" %}
Build to static HTML and host it anywhere — a full-bleed hero with a readable scrim.
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', variant='image', image='/landscape.jpg', alt='',
          title='Deploy anywhere', cta='Ship it', href='/docs/#quickstart',
          children='Build to static HTML and host it anywhere — a full-bleed hero '
                   'with a readable scrim.')
```

Gradient and glass

`variant="gradient"` paints the card with a gradient (derived from the theme accent, or set explicitly with `gradient="from,to,deg"`):

✨Pro plan

Priority support and unlimited seats.

Source: Markdown

```
{% card variant="gradient" gradient="indigo,cyan,135" icon="sparkles" title="Pro plan"
cta="Upgrade" href="/docs/#quickstart" %}
Priority support and unlimited seats.
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', variant='gradient', gradient='indigo,cyan,135',
icon='sparkles',
          title='Pro plan', cta='Upgrade', href='/docs/#quickstart',
          children='Priority support and unlimited seats.')
```

`variant="glass"` is a translucent, frosted surface — its **backdrop blur** shows whatever sits behind it, so it's best over a colorful background or the page's background image (shown here over a background image so the blur is visible):

Frosted glass

A translucent surface with a backdrop blur.

Source: Markdown

```
{% card variant="glass" icon="diamond" title="Frosted glass" %}
A translucent surface with a backdrop blur.
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', variant='glass', icon='diamond', title='Frosted glass',
          children='A translucent surface with a backdrop blur.')
```

Stat cards and accent bars

`variant="stat"` is a centered big-number layout for KPIs (the `title` is the number, the `subtitle` is the label). `accentBar="top"` (or `"left"`) adds a gradient accent edge:

99.99%

<40ms

4.2M

Source: Markdown

```
{% cardGrid cols=3 %}
{% card variant="stat" title="99.99%" subtitle="Uptime" icon="activity" accent="teal"
align="center" accentBar="top" %}{% endCard %}
{% card variant="stat" title="<40ms" subtitle="p99 latency" icon="gauge" accent="blue"
```

```
align="center" accentBar="top" %}{% endCard %}
{% card variant="stat" title="4.2M" subtitle="Builds / day" icon="rocket" accent="grape"
align="center" accentBar="top" %}{% endCard %}
{% endCardGrid %}
```

Source: Python

```
stats = [('99.99%', 'Uptime', 'activity', 'teal'),
         ('<40ms', 'p99 latency', 'gauge', 'blue'),
         ('4.2M', 'Builds / day', 'rocket', 'grape')]
cards = ''
for title, subtitle, icon, accent in stats:
    cards += component('aardvark', 'card', variant='stat', title=title, subtitle=subtitle,
                       icon=icon, accent=accent, align='center', accentBar='top')
page.print(component('aardvark', 'cardGrid', cols=3, children=cards))
```

The grid

`{% cardGrid %}` lays cards out in a responsive grid that collapses to one column on mobile. `cols=N` sets the desktop count and steps down on its own — two columns on a tablet, one on a phone — so the cards wrap instead of shrinking. Use `colsBase`/`colsSm`/`colsMd`/`colsLg` when you want to pin each breakpoint yourself (`colsBase=N` alone gives a genuinely fixed `N` at every width); a `cols=N` written alongside them still means the desktop count, and any breakpoint you pin wins. `spacing` sets the gap.

From a loop or a data file

A `{% card %}` tag is expanded as the page is scanned, so it can't be written inside a `{% %}` `for` loop. To generate cards from a data file (or any list), use the Python form — `component('aardvark', 'card', …)` and `component('aardvark', 'cardGrid', …)` — which return the same markup the tags do (there's one implementation behind both). Keyword arguments are the same attributes you'd write on the tag; `children` is the body:

Source: Python

```
{%
cards = ''
for item in data.products.items:
    cards += component('aardvark', 'card', icon=item.icon, title=item.name,
subtitle=item.blurb,
                       badge='$' + str(item.price), cta='Buy now')
page.print(component('aardvark', 'cardGrid', cols=3, children=cards))
%}
```

The component libraries page builds a small storefront exactly this way — each card wraps a live Stripe element, all driven from `data/products.yaml`.

With other components

A card body is full Markdown, so it holds any other component — here a {% badge %} and an {% icon %} ride inside the body, and an {% image %} sits beneath the grid:

Status

All systems [Operational]

Release

v2.0 [New] ships today.

Source: Markdown

```
{% cardGrid colsBase=1 colsSm=2 %}
{% card title="Status" icon="server" accent="teal" %}
All systems {% icon "circle-check" color="green" %} {% badge color='green' variant='light'
%}Operational{% endBadge %}
{% endCard %}
{% card title="Release" icon="package" accent="grape" %}
v2.0 {% badge color='grape' %}New{% endBadge %} ships today.
{% endCard %}
{% endCardGrid %}
```

Source: Python

```
body1 = ("All systems " + component('aardvark', 'icon', 'circle-check', color='green')
         + " " + component('aardvark', 'badge', color='green', variant='light',
children='Operational'))
body2 = "v2.0 " + component('aardvark', 'badge', color='grape', children='New') + " ships
today."
cards = component('aardvark', 'card', title='Status', icon='server', accent='teal',
children=body1)
cards += component('aardvark', 'card', title='Release', icon='package', accent='grape',
children=body2)
page.print(component('aardvark', 'cardGrid', colsBase=1, colsSm=2, children=cards))
```

Attributes

`{% card %}`

Set what you need, omit the rest.

Attribute	Valid values	Description
`title`	string	Card heading. Also the accessible link name when `href` is set — give a link card a `title` so screen readers announce it meaningfully (a title-less link card falls back to “Open”).
`subtitle`	string	A dimmed line under the title (the label, for `stat` cards).
(body)	Markdown	Content below the heading. Leave empty for a bare/stat card.
`href` / `url`	URL	Make the **whole card** a link. External URLs open in a new tab with an icon, and hovering the card shows the destination domain as a tooltip. (`url` is used only when `href` is absent.)
`icon`	a Tabler name (`rocket`, `brand-github`), a Font Awesome class (`fa-solid fa-rocket`), an emoji, an image URL, or inline `<svg>`	The icon chip.
`image`	image URL	Cover image (top) — or the full background when `variant="image"`.
`alt`	string	Alt text for the image (use `alt=""` for a decorative image).
`variant`	`elevated` (default), `gradient`, `glass`, `image`, `stat`, `plain`	Card style. `plain` is a bare Mantine card — no shadow or hover lift.
`badge`	string	One badge label (any text, including a comma).
`badges`	comma-separated string	Several badge labels (so an individual label can’t contain a comma — use `badge` for one that does).
`badgeColor`	any theme/CSS color	Badge color.

`badgeVariant`	`light` (default), `filled`, `outline`, `dot`, `transparent`, `white`, `gradient`	Mantine `Badge` variant.
`accent` / `color`	a Mantine palette name, a theme color, or any CSS color	Accent color for the icon chip, accent bar, hover border, and CTA.
`gradient`	`"from,to,deg"`, a single color (`"indigo"`), or set via `gradientFrom`/`gradientTo`/`gradientDeg`	Gradient stops for `variant="gradient"`.
`accentBar`	`top`, `left`	A gradient accent edge.
`cta`	string	A call-to-action **button** at the bottom.
`ctaVariant`	`link`	Render the CTA as a text link with an arrow instead of a button.
`buttonVariant`	`filled`, `light`, `outline`, `subtle`, `white`, `gradient`, …	Mantine `Button` variant for the CTA button. Defaults to `white` on photo/gradient cards, `light` otherwise.
`align`	`left` (default), `center`	Content alignment (`center` suits icon-led and stat cards).
`shadow`	Mantine shadow token (`xs`–`xl`)	Card shadow.
`radius`	`xs`–`xl` or any CSS value	Corner rounding.
`withBorder`	bool flag	Draw a border.
`padding`	Mantine spacing token or CSS size	Inner padding.
`sheen`	bool flag (`sheen=false`)	Turn off the animated hover sheen on a link card.
`id`	string	An HTML `id`.
`onclick`	JS expression	A JS click handler (a raw HTML attribute).

`{% cardGrid %}`

Attribute	Valid values	Description
`cols`	integer	Columns on desktop (collapses to 2 on tablet, 1 on mobile). Default `{base:1, sm:2, lg:3}`.
`colsBase` `colsSm` `colsMd` `colsLg` `colsXl`	integer	Per-breakpoint column counts. On their own they define the grid outright (no step-down); alongside `cols` they override just those breakpoints and the rest of the step-down stays.
`spacing`	Mantine spacing token or CSS size	Horizontal gap between cards.
`verticalSpacing`	Mantine spacing token or CSS size	Vertical gap between cards.

This tag ships as a built-in; when you need something different, define your **own** custom component and call it the same way.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Card"]` — or through the Mantine Styles API classes (`.mantine-Card-root` and its inner parts):

```
/* Every rendered Card carries this island marker */
[data-aardvark-island="Card"] { }

/* Mantine Styles API class on the root element */
.mantine-Card-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — onto the rendered card. `card` also exposes a dedicated `onclick` shortcut for the common click case; in Python it rides this same channel as `attr={'onclick': …}`.

What is Aardvark?

A documentation-focused static site generator.

Source: Markdown

```
{% card title="What is Aardvark?" onclick="alert(this.innerText)" %}
A documentation-focused static site generator.
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', title='What is Aardvark?',
          children='A documentation-focused static site generator.',
          attr={'onclick': 'alert(this.innerText)'})
```

For any other attribute — `data-*`, ARIA, analytics hooks — pass the `attr={…}` dict directly:

Pricing

Three tiers, no surprises.

Source: Markdown

```
{% card title="Pricing" attr={'data-analytics': 'pricing-card', 'aria-label': 'Pricing
card'} %}
Three tiers, no surprises.
{% endCard %}
```

Source: Python

```
component('aardvark', 'card', title='Pricing', children='Three tiers, no surprises.',
          attr={'data-analytics': 'pricing-card', 'aria-label': 'Pricing card'})
```

Carousel

A **slides carousel**: a strip of slides the reader swipes, drags, or steps through with the prev/next controls and indicator dots. Put one `{% slide %}` block per slide inside a `{% carousel %}`. It's embla-backed and measures the live container on mount, so it renders in the browser and hydrates into an interactive island.

Use it as `{% carousel %}…{% endCarousel %}` in Markdown (with nested `{% slide %}` blocks), or call it from Python logic (loops, snippets) via `component('aardvark', 'carousel', …)` and `component('aardvark', 'slide', …)`.

A basic carousel

Each `{% slide %}` body is the slide content (rendered as Markdown). Set `slideSize` and `slideGap` to fit more than one slide in view at a time.

Slide one — the first panel in the strip.

Slide two — drag, swipe, or use the arrows.

Slide three — `loop` wraps back to the start.

Source: Markdown

```
{% carousel slideSize='70%' slideGap='md' withIndicators=true loop=true %}
{% slide %}
**Slide one** — the first panel in the strip.
{% endSlide %}
{% slide %}
**Slide two** — drag, swipe, or use the arrows.
{% endSlide %}
{% slide %}
**Slide three** — `loop` wraps back to the start.
{% endSlide %}
{% endCarousel %}
```

Source: Python

```
component('aardvark', 'carousel', slideSize='70%', slideGap='md',
          withIndicators=True, loop=True,
          children=(component('aardvark', 'slide', children='**Slide one**')
                    + component('aardvark', 'slide', children='**Slide two**')
                    + component('aardvark', 'slide', children='**Slide three**')))
```

Orientation and sizing

Set `orientation='vertical'` for a top-to-bottom carousel — vertical needs a `height`. Use `controlSize` and `controlsOffset` to tune the arrow buttons, and turn `withControls` off for a swipe-only carousel.

Top — a vertical strip stacks its slides.

Middle — scroll with the arrows or drag.

Bottom — `height` is required when vertical.

Source: Markdown

```
{% carousel orientation='vertical' height='180' slideGap='sm' withControls=true %}
{% slide %}
**Top** — a vertical strip stacks its slides.
{% endSlide %}
{% slide %}
**Middle** — scroll with the arrows or drag.
{% endSlide %}
{% slide %}
**Bottom** — `height` is required when vertical.
{% endSlide %}
{% endCarousel %}
```

Attributes

Set these on `{% carousel %}`; omit any to take its Mantine default. `{% slide %}` takes no attributes of its own — its body is the slide content.

Attribute	Valid values	Description
`slideSize`	percentage (`70%`) or a number	Slide width relative to the viewport.
`slideGap`	`xs`–`xl` or a CSS length	Gap between slides.
`height`	a CSS height	Slides container height; required for `vertical`.
`h`	`xs`–`xl` or a CSS length	Mantine height shorthand on the root box.
`orientation`	`horizontal`, `vertical`	Scroll direction.

`controlSize`	a CSS width (e.g. `26`)	Size of the prev/next controls.
`controlsOffset`	`xs`–`xl` or a CSS value	Distance of the controls from the edges.
`initialSlide`	integer	Index of the slide shown first.
`loop`	bool (`true` / `false`)	Wrap from the last slide back to the first.
`withControls`	bool (`true` / `false`)	Show the prev/next arrows (default `true`).
`withIndicators`	bool (`true` / `false`)	Show the indicator dots (default `false`).
`withKeyboardEvents`	bool (`true` / `false`)	Arrow keys switch slides (default `true`).

CSS Selectors

The carousel hydrates from a server-rendered island wrapper, so you can target it before hydration with the Aardvark island attributes, then reach into the Mantine parts once it mounts. The island name stays the bare component (`Carousel` / `CarouselSlide`); the library key rides a separate `data-aardvark-lib`:

```
/* The carousel and slide island wrappers (any library). */
[data-aardvark-island="Carousel"] { }
[data-aardvark-island="CarouselSlide"] { }

/* Scope to this library only (in case another library exposes a same-named component). */
[data-aardvark-lib="carousel"][data-aardvark-island="Carousel"] { }

/* The mounted Mantine parts, once the island hydrates. */
.mantine-Carousel-root { }       /* the carousel root */
.mantine-Carousel-slide { }      /* each slide */
.mantine-Carousel-controls { }   /* the prev/next control group */
.mantine-Carousel-indicators { } /* the indicator dots */
```

Injecting Attributes

Pass `attr={...}` to forward raw HTML attributes — a `data-*` hook, an `id`, an event handler — straight onto the carousel's rendered element. In Markdown the dict is written inline; in Python it's the `attr=` keyword.

Tap the carousel — the click handler reports its text content.

Second slide — the handler is on the carousel root, not the slides.

Source: Markdown

```
{% carousel withIndicators=true attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
{% slide %}
**Tap the carousel** — the click handler reports its text content.
{% endSlide %}
{% slide %}
**Second slide** — the handler is on the carousel root, not the slides.
{% endSlide %}
{% endCarousel %}
```

Source: Python

```
component('aardvark', 'carousel', withIndicators=True, attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''},
          children=(component('aardvark', 'slide',
                              children='**Tap the carousel** — the click handler reports its
text content.')
                    + component('aardvark', 'slide',
                                children='**Second slide** — the handler is on the carousel
root, not the slides.')))
```

Whatever JavaScript you put in `onclick` ships straight to readers' browsers; you can lock that down site-wide with the `attrPolicy` block in `aardvark.config.yaml`.

ColorSwatch

A square swatch that displays one color — handy in palettes, legends, and design docs. The `color` attribute takes any CSS color value: a hex, an `rgb()`/`rgba()`, or a named color. An optional block body renders **inside** the swatch.

Use it as `{% colorswatch %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'colorswatch', …)`.

Demonstrations

Color values

`color` accepts hex, `rgba()`, and named colors:

Source: Markdown

```
{% colorswatch color='#7048e8' %}
{% colorswatch color='rgba(112, 72, 232, 0.5)' %}
{% colorswatch color='#e64980' %}
{% colorswatch color='#12b886' %}
{% colorswatch color='#fab005' %}
```

Source: Python

```
component('aardvark', 'colorswatch', color='#7048e8')
component('aardvark', 'colorswatch', color='rgba(112, 72, 232, 0.5)')
component('aardvark', 'colorswatch', color='#e64980')
component('aardvark', 'colorswatch', color='#12b886')
component('aardvark', 'colorswatch', color='#fab005')
```

Size, radius, and shadow

`size` sets the width and height (any CSS value; numbers are rem). `radius` squares off the corners. `withShadow` is on by default — pass `withShadow=false` to drop the inner shadow.

Source: Markdown

```
{% colorswatch color='#7048e8' size='1rem' %}
{% colorswatch color='#7048e8' size='2.5rem' %}
{% colorswatch color='#7048e8' radius='sm' %}
{% colorswatch color='#7048e8' withShadow=false %}
```

Source: Python

```
component('aardvark', 'colorswatch', color='#7048e8', size='1rem')
component('aardvark', 'colorswatch', color='#7048e8', size='2.5rem')
```

```
component('aardvark', 'colorswatch', color='#7048e8', radius='sm')
component('aardvark', 'colorswatch', color='#7048e8', withShadow=False)
```

Content inside the swatch

An optional block body renders inside the swatch — for example, a check to mark the selected color:

✓

Source: Markdown

```
{% colorswatch color='#12b886' size='2rem' %}{% icon 'check' %}{% endColorswatch %}
```

Source: Python

```
component('aardvark', 'colorswatch', color='#12b886', size='2rem',
          children=component('aardvark', 'icon', 'check'))
```

With other components

Swatches line up well inside a `{% group %}` as a legend or palette:

Source: Markdown

```
{% group gap='xs' %}
{% colorswatch color='#7048e8' %}
{% colorswatch color='#e64980' %}
{% colorswatch color='#12b886' %}
{% colorswatch color='#fab005' %}
{% endGroup %}
```

Source: Python

```
swatches = ''.join(
    component('aardvark', 'colorswatch', color=c)
    for c in ('#7048e8', '#e64980', '#12b886', '#fab005')
)
component('aardvark', 'group', gap='xs', children=swatches)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description

`color`	any CSS color value — `#fff`, `rgba(...)`, a named color (**required**)	The color the swatch displays.
`size`	any CSS value (numbers are rem)	The swatch's width and height.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	Corner rounding.
`withShadow`	bool flag (default `true`)	Inner box-shadow. Set `withShadow=false` to remove it.
body	Markdown / components	Content rendered inside the swatch (e.g. a check icon).

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="ColorSwatch"]` — or through the Mantine Styles API classes (`.mantine-ColorSwatch-root` and its inner parts):

```
/* Every rendered ColorSwatch carries this island marker */
[data-aardvark-island="ColorSwatch"] { }

/* Mantine Styles API class on the root element */
.mantine-ColorSwatch-root { }
.mantine-ColorSwatch-colorOverlay { }
.mantine-ColorSwatch-shadowOverlay { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% colorswatch color='#7048e8' attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'colorswatch', color='#7048e8', attr={'onclick': '''
const value = this.tagName;
console.log('attr demo value:', value);
```

```
alert(value);
'''})
```

DonutChart

A **donut chart** for part-to-whole breakdowns. Give it the segments in `data` (a JSON array of `{name, value, color}`), size it with `size` / `thickness`, and optionally show a center `chartLabel`. Renders in the browser and hydrates into an interactive island.

Use it as `{% donutchart %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'donutchart', …)`.

A basic donut chart

Source: Markdown

```
{% donutchart size=180 thickness=28 withLabels=true chartLabel='Traffic' data='[
  {"name":"Search","value":48,"color":"indigo.6"},
  {"name":"Direct","value":27,"color":"teal.6"},
  {"name":"Referral","value":25,"color":"orange.6"}
]' %}
```

Source: Python

```
component('aardvark', 'donutchart', size=180, thickness=28, withLabels=True,
chartLabel='Traffic',
          data='[{"name":"Search","value":48,"color":"indigo.6"}, …]')
```

Attributes

Attribute	Valid values	Description
`data`	JSON array of `{name, value, color}`	The donut segments.
`size`	integer (px, default `160`)	Outer chart size.
`thickness`	integer (px, default `20`)	Ring thickness.
`chartLabel`	string	Text shown in the center.
`withLabels`	bool (`true` / `false`)	Label each segment.
`withTooltip`	bool (default `true`)	Show the hover tooltip.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="DonutChart"]` (or the more specific `[data-aardvark-lib="charts"][data-aardvark-island="DonutChart"]` when several libraries share the page) — or through the Mantine Styles API classes (`.mantine-DonutChart-root` and its inner parts):

```
/* Every rendered DonutChart carries this island marker */
[data-aardvark-island="DonutChart"] { }

/* Scope by library + island when you have several libraries in play */
[data-aardvark-lib="charts"][data-aardvark-island="DonutChart"] { }

/* Mantine Styles API class on the root element */
.mantine-DonutChart-root { }
.mantine-DonutChart-label { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% donutchart size=180 thickness=28 withLabels=true chartLabel='Traffic' data='[
  {"name":"Search","value":48,"color":"indigo.6"},
  {"name":"Direct","value":27,"color":"teal.6"},
  {"name":"Referral","value":25,"color":"orange.6"}
]' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'donutchart', size=180, thickness=28, withLabels=True,
chartLabel='Traffic',
          data='''[
  {"name":"Search","value":48,"color":"indigo.6"},
  {"name":"Direct","value":27,"color":"teal.6"},
  {"name":"Referral","value":25,"color":"orange.6"}
]''', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Icon

A **built-in** inline tag for dropping an icon inside text — a sentence, a list item, or a heading. It renders a Mantine ThemeIcon, so it carries the whole ThemeIcon surface — `variant`, `color`, `size`, `radius`, `gradient` — while still flowing inline. By default it's a bare glyph that **inherits the surrounding text's size and color**; add a `variant`/`color` to turn it into a Mantine icon chip. Use it as `{% icon "rocket" %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'icon', 'rocket', …)`.

The first argument is the icon, quoted. It can be a **Tabler name** (a bare lowercase name like `rocket` or `brand-github`), a **Font Awesome class** (`fa-solid fa-rocket`, `fab fa-github`), a path/URL to an **SVG or image file** (`/icons/folder.svg`), or an **emoji** (`⍰`).

Demonstrations

Inline glyphs

A bare name renders an inline glyph that inherits the surrounding text's size and color:

Launch it when you're ready, then star it on GitHub.

Source: Markdown

```
Launch it {% icon "rocket" %} when you're ready, then star it {% icon "star" %} on GitHub {% icon "brand-github" %}.
```

Source: Python

```
page.print("Launch it " + component('aardvark', 'icon', 'rocket')
           + " when you're ready, then star it " + component('aardvark', 'icon', 'star')
           + " on GitHub " + component('aardvark', 'icon', 'brand-github') + ".")
```

Variants and colors

Pass any Mantine ThemeIcon `variant` — `filled`, `light`, `outline`, `transparent` (the default), `white`, `default`, or `gradient` — with a `color` to turn the bare glyph into a chip. A gradient `variant` reads `gradientFrom`/`gradientTo`/`gradientDeg`:

Source: Markdown

```
{% icon "rocket" variant="filled" color="blue" %}
{% icon "heart" variant="light" color="pink" %}
{% icon "star" variant="outline" color="yellow" %}
{% icon "bell" variant="white" color="grape" %}
```

```
{% icon "flame" variant="gradient" gradientFrom="orange" gradientTo="red" %}
```

Source: Python

```
component('aardvark', 'icon', 'rocket', variant='filled', color='blue')
component('aardvark', 'icon', 'heart', variant='light', color='pink')
component('aardvark', 'icon', 'star', variant='outline', color='yellow')
component('aardvark', 'icon', 'bell', variant='white', color='grape')
component('aardvark', 'icon', 'flame', variant='gradient', gradientFrom='orange',
gradientTo='red')
```

`color` is a Mantine color name (`blue`, `pink`, …) or any CSS color. `autoContrast` flips the glyph between light/dark for legibility on its background.

Size and radius

`size` takes a Mantine size (`xs`–`xl`) or a number of pixels; `radius` takes a Mantine radius (`xs`–`xl`) or a number:

🚀🚀🚀🚀

Source: Markdown

```
{% icon "rocket" variant="filled" color="blue" size="xs" %}
{% icon "rocket" variant="filled" color="blue" size="sm" %}
{% icon "rocket" variant="filled" color="blue" size="lg" %}
{% icon "rocket" variant="filled" color="blue" size="xl" radius="xl" %}
```

Source: Python

```
for s in ('xs', 'sm', 'lg', 'xl'):
    page.print(component('aardvark', 'icon', 'rocket', variant='filled', color='blue',
size=s))
```

With **no `size`**, the icon is `1em` and tracks the surrounding text — so a bare icon grows with whatever it sits in, including headings:

Ship faster 🚀

Built for teams 👥

Nudging and spacing

A glyph sometimes sits a hair high or low for your text, or you want a little space around it. Use Mantine's margin/padding style props — `mt`/`mb`/`ml`/`mr` (or the `mx`/`my`/`m` shorthands) for margin, and `pt`/`pb`/… (or `px`/`py`/`p`) for padding. A bare number is **pixels**, and negatives nudge:

Lift it ☆ up, drop it ☆ down, or space it ☆ out.

Source: Markdown

```
Lift it {% icon "star" mt=-2 %} up, drop it {% icon "star" mt=2 %} down, or space it {% icon
"star" mx=8 %} out.
```

Source: Python

```
page.print("Lift it " + component('aardvark', 'icon', 'star', mt=-2)
           + " up, drop it " + component('aardvark', 'icon', 'star', mt=2)
           + " down, or space it " + component('aardvark', 'icon', 'star', mx=8) + " out.")
```

A named Mantine spacing works too (`mt="xs"`), and these apply to every variant.

Tooltip and accessible name

Give an icon a `label` and hovering (or focusing) it shows a Mantine **Tooltip** with that text. The label is also the icon's accessible name (it sets `aria-label`) and makes the icon keyboard-focusable, so the tooltip works without a mouse and a screen-reader user hears the name too. Use it when a **standalone** icon is the meaning and nothing adjacent already says it:

Source: Markdown

```
{% icon "brand-github" label="View on GitHub" %} {% icon "rocket" variant="filled"
color="blue" label="Deploy" %}
```

Source: Python

```
component('aardvark', 'icon', 'brand-github', label='View on GitHub')
component('aardvark', 'icon', 'rocket', variant='filled', color='blue', label='Deploy')
```

Icons without a `label` are decorative — they're hidden from assistive tech (`aria-hidden`), and the surrounding text carries the meaning.

Don't `label` an icon that's the content of a link or button. A labelled icon is its own keyboard focus stop, so wrapping one in an `<a>`/`<button>` makes a second tab stop and a doubled screen-reader announcement. Leave the icon decorative (no `label`) and put the accessible name on the **outer** element instead — name the link, not the icon:

Source: Markdown

```
<a href="https://github.com/…" aria-label="View on GitHub">{% icon "brand-github" %}</a>
```

Tabler, Font Awesome, image, and emoji

A **bare name** is a Tabler icon — there are over 5,800. Tabler ships an **outline** style by default and a **filled** style for many icons; add the `filled` flag for the filled variant. A **Font Awesome** glyph needs its **full class** (the style class `fa-solid`/`fa-regular`/`fab`/… plus the icon class) and the FA stylesheet (`theme.fontawesome: true` in `aardvark.config.yaml`, which this site sets). A path or URL ending in `.svg`/`.png`/`.webp`/… renders as an inline `<img>`, and a non-ASCII glyph renders as an emoji:

♡outline, ♥ filled — — open the 🗀 folder, or just ship it.

Source: Markdown

```
{% icon "heart" %} outline, {% icon "heart" filled %} filled
{% icon "fa-solid fa-rocket" %} {% icon "fab fa-github" %} {% icon "fa-regular fa-star" %}
{% icon "/icons/folder.svg" %} {% icon "🗀" %}
```

Source: Python

```
component('aardvark', 'icon', 'heart')             # Tabler outline
component('aardvark', 'icon', 'heart', filled=True) # Tabler filled
component('aardvark', 'icon', 'fa-solid fa-rocket') # Font Awesome
component('aardvark', 'icon', '/icons/folder.svg')  # image
component('aardvark', 'icon', '🗀')                 # emoji
```

A bare `rocket` is the **Tabler** `rocket`, so Font Awesome always needs its `fa-…` class. Image and emoji icons keep their own colors — the `color` override tints a Tabler or Font Awesome glyph only.

With other components

An icon flows inline inside any other component — here it leads a {% badge %} and labels a {% card %}:

[Verified] — status ⓥ

ϟ Fast builds

The icon chip up top is the same engine — `{% card %}` takes an `icon=`.

Source: Markdown

```
{% badge color='green' variant='light' leftSection='✓' %}Verified{% endBadge %} — status {%
icon "circle-check" color="green" label="All systems go" %}

{% card title="Fast builds" icon="bolt" accent="yellow" %}
The icon chip up top is the same engine — {% card %} takes an icon=.
{% endCard %}
```

Source: Python

```
page.print(component('aardvark', 'badge', color='green', variant='light',
                     leftSection='✓', children='Verified')
           + " — status "
           + component('aardvark', 'icon', 'circle-check', color='green', label='All systems
go'))
page.print(component('aardvark', 'card', title='Fast builds', icon='bolt', accent='yellow',
                     children='The icon chip up top is the same engine.'))
```

Loading and the CDN

The `{% icon %}` tag is a Mantine island, so it needs the islands bundle (which this site builds). **Tabler** glyphs add one more step: the first time a page uses a Tabler icon, Aardvark fetches its SVG from a pinned jsDelivr CDN (`cdn.jsdelivr.net`), caches it per name, and injects it inline — only the icons a page actually uses are fetched.

Because that fetch is client-side, a **Tabler** glyph renders only after the page's JavaScript runs: the build's server-side prerender bakes an empty reserved box (no layout shift), and the glyph pops in on mount. **Font Awesome, image, and emoji** icons are baked into the prerendered HTML and show with no JavaScript. So a site that must render icons without JS (or before hydration) should prefer Font Awesome or image icons over Tabler.

Navigation `icon:` and `heading-icon:` values are latency-sensitive page chrome, so Aardvark special-cases them: their SVGs are baked into the initial HTML and paint immediately — from the outline set bundled with Aardvark (so it works even with no local `@tabler/icons` install, e.g. a `pip`/binary site), or from the installed package when it matches a pinned `theme.iconVersion`. A custom `theme.iconCdn` stays authoritative — navigation keeps its runtime loader and `vark build` emits no warning. A pinned `theme.iconVersion` that's neither bundled nor installed also falls back to the runtime loader, and there `vark build` warns which glyphs it couldn't bake.

If your site sets a **Content-Security-Policy**, allow the CDN origin in `connect-src` (the fetch). To self-host the icons or pin a different release, set one of these in `aardvark.config.yaml`:

- `theme.iconCdn` — a full `http(s)` base that serves `<base>/outline/<name>.svg` (and `<base>/filled/<name>.svg`);
- `theme.iconVersion` — a bare `X.Y.Z` to pin the default jsDelivr package version.

Attributes

Attribute	Valid values	Description
(first arg)	a **Tabler** name (`rocket`, `brand-github`), a **Font Awesome** class (`fa-solid fa-rocket`), an **image** path/URL (`/icons/logo.svg`), or an **emoji**	The icon. Must be quoted. A bare name is Tabler — Font Awesome needs its `fa-…` class.
`variant`	`filled`, `light`, `outline`, `transparent` (default), `white`, `default`, `gradient`	Mantine ThemeIcon variant.
`color`	a Mantine color name or any CSS color	Tints the glyph (and the chip for non-transparent variants); image/emoji keep their own colors.
`size`	a Mantine size (`xs`–`xl`) or a number of pixels	Icon size. With no `size`, the icon is `1em` and tracks the surrounding text.
`radius`	a Mantine radius (`xs`–`xl`) or a number	The chip's corner rounding.
`mt` `mb` `ml` `mr` `mx` `my` `m` / `pt` … `p`	a bare number (pixels, negatives nudge) or a Mantine size (`mx="xs"`)	Mantine margin/padding style props to nudge/space a glyph.
`gradientFrom` / `gradientTo` / `gradientDeg`	colors / integer degrees	For `variant="gradient"`, compose the `{from, to, deg}`. Omit to use the theme's default gradient.
`autoContrast`	bool flag	Flip the glyph to light/dark for legibility against its background.
`filled`	bool flag	Use Tabler's **filled** variant instead of the default outline (Tabler icons only).

`label`	string	Show a Mantine **Tooltip** on hover/focus, name the icon for assistive tech (`aria-label`), and make it keyboard-focusable. For **standalone** icons only — for an icon inside a link/button, leave it decorative and name the outer element instead. Omit it for decorative icons, which are hidden from assistive tech.

A bare name (no `fa-…` class, no file extension, ASCII) is read as a Tabler icon — if it isn't a real Tabler name, the loader logs a console warning and leaves the slot empty, so check the name at tabler.io/icons. Keep `{% icon %}` on the same line as adjacent text: an icon alone on its own line is treated as a block, not an inline glyph.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Icon"]` — or through the Mantine Styles API classes (`.mantine-ThemeIcon-root` and its inner parts):

```
/* Every rendered Icon carries this island marker */
[data-aardvark-island="Icon"] { }

/* Mantine Styles API class on the root element */
.mantine-ThemeIcon-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — `id`, `data-*`, ARIA, analytics hooks, or an inline event handler — onto the rendered icon. (For the icon's own surface — `variant`, `color`, `size`, `radius`, gradient, and `label` for its accessible name and hover tooltip — use the documented props above.)

🚀

Source: Markdown

```
{% icon "rocket" variant="filled" color="blue" label="Deploy" attr={'data-analytics':
'deploy-icon'} %}
```

Source: Python

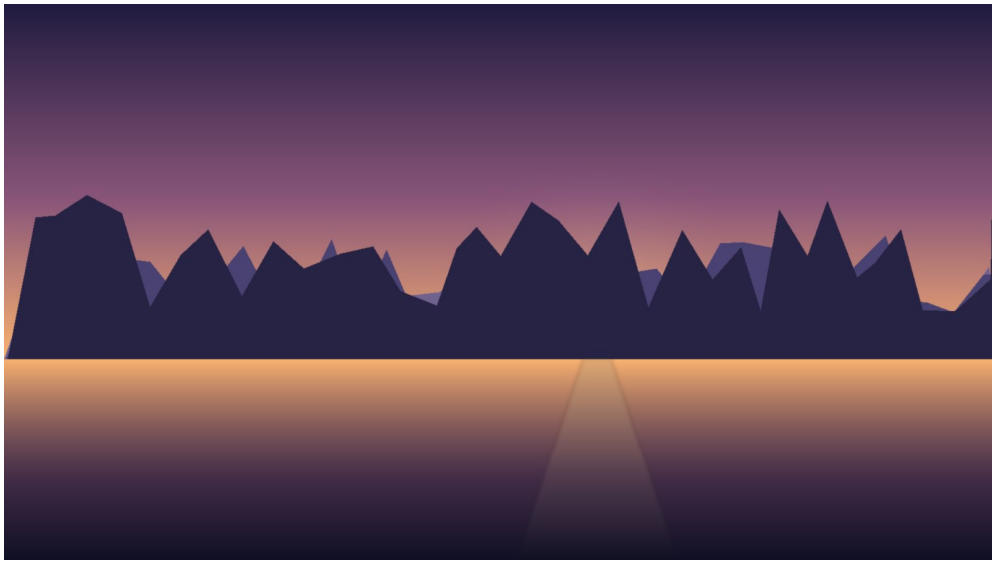
```
component('aardvark', 'icon', 'rocket', variant='filled', color='blue', label='Deploy',
          attr={'data-analytics': 'deploy-icon'})
```

Image

Images render through Mantine's `Image` component with two extras Aardvark adds on top: they fill the content width **without** being upscaled past their natural size, and — when there's more detail to reveal — clicking one opens it in a **dimmed full-image lightbox**. A plain Markdown image on its own line gets this automatically; use the `{% image %}` tag when you need options. Use it as `{% image %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'image', …)`.

Plain Markdown images

A normal Markdown image on its own line gets the treatment automatically — write Markdown as usual. A large image fills the content column; a small one stays its natural size (no blurry upscaling). The Markdown **title** (the quoted text after the URL) shows as a hover **tooltip**, falling back to the **alt text** when there's no title. Images written inside a sentence stay plain inline images.



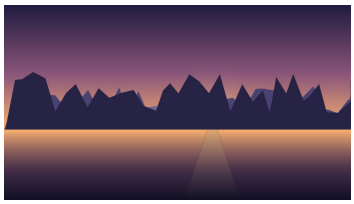
Source: Markdown

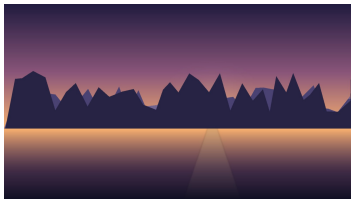
```
![A still mountain lake at dusk](/landscape.jpg "Hover me — dusk over the lake")
```

Demonstrations

Sizing and alignment

`size` scales the image to a percentage of the content width (aspect ratio preserved), and `align` (`left`, `center`, or `right`) positions a scaled image:





Source: Markdown

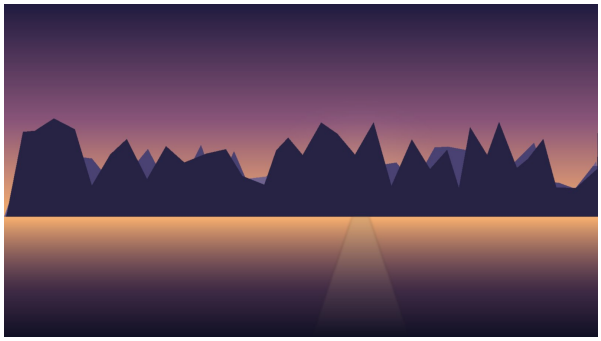
```
{% image src='/landscape.jpg' alt='Mountains at dusk over a still lake' size='35%'
align='left' %}
{% image src='/landscape.jpg' alt='Mountains at dusk over a still lake' size='35%'
align='center' %}
```

Source: Python

```
component('aardvark', 'image', src='/landscape.jpg',
          alt='Mountains at dusk over a still lake', size='35%', align='left')
component('aardvark', 'image', src='/landscape.jpg',
          alt='Mountains at dusk over a still lake', size='35%', align='center')
```

Captions

`caption` renders a visible figure caption below the image — opt-in, so plain Markdown images never grow one. The figure shrinks to the image, so the caption is exactly as wide as what it describes:



Source: Markdown

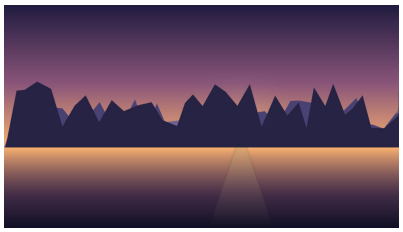
```
{% image src='/landscape.jpg' alt='A still mountain lake at dusk' caption='Figure 1 — dusk
over the lake' size='60%' align='center' %}
```

Source: Python

```
component('aardvark', 'image', src='/landscape.jpg',
          alt='A still mountain lake at dusk',
          caption='Figure 1 — dusk over the lake', size='60%', align='center')
```

Disabling zoom

Zoom is offered only while the image is shown smaller than its natural size — an image already at full size stays static (a lightbox would have nothing more to reveal). Set `zoom=false` to disable the lightbox entirely, even while scaled down (decorative art, or a diagram you don't want enlarged):



Source: Markdown

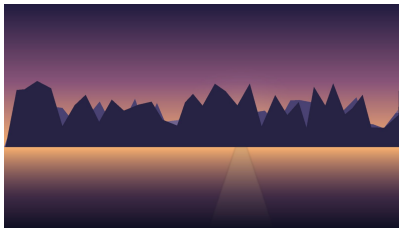
```
{% image src='/landscape.jpg' alt='Mountains at dusk over a still lake' size='40%'
zoom=false %}
```

Source: Python

```
component('aardvark', 'image', src='/landscape.jpg',
          alt='Mountains at dusk over a still lake', size='40%', zoom=False)
```

Radius, fit, and framing

`radius` rounds the corners (`xs`–`xl` or any CSS value), `fit` sets `object-fit`, and the visual props `bd`/`bg`/`opacity` plus the spacing/sizing props frame the image:



Source: Markdown

```
{% image src='/landscape.jpg' alt='Mountains at dusk over a still lake' size='40%'
radius='lg' bd='2px solid var(--mantine-color-grape-5)' %}
```

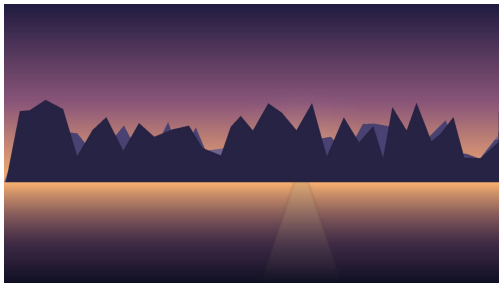
Source: Python

```
component('aardvark', 'image', src='/landscape.jpg',
          alt='Mountains at dusk over a still lake', size='40%',
          radius='lg', bd='2px solid var(--mantine-color-grape-5)')
```

With other components

An image sits inside any other component — here a cover image leads a {% card %} (via its own `image` attribute), and a standalone `{% image %}` sits beside a {% badge %} caption:

[Figure 2]



Source: Markdown

```
{% badge color='blue' variant='light' %}Figure 2{% endBadge %}

{% image src='/landscape.jpg' alt='A still mountain lake at dusk' size='50%' align='center'
caption='A lake at dusk' %}
```

Source: Python

```
page.print(component('aardvark', 'badge', color='blue', variant='light', children='Figure
2'))
page.print(component('aardvark', 'image', src='/landscape.jpg',
                     alt='A still mountain lake at dusk', size='50%',
                     align='center', caption='A lake at dusk'))
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`src`	image URL	The image (required).
`alt`	string	Alternative text. Shows as a hover tooltip and names the lightbox dialog for screen readers. Use `alt=""` for a purely decorative image.
`caption`	string	Visible figure caption shown below the image. Opt-in — omit it for no caption.

`size`	percentage, e.g. `50%`	Scale the image to a percentage of the content width. Aspect ratio is preserved. Defaults to natural size, capped to the column.
`align`	`left` (default), `center`, `right`	Positions a scaled image.
`zoom`	bool flag (`true` default / `false`)	`true` opens a lightbox on click while the image is scaled down (an image at full size isn't clickable). `false` disables it entirely.
`fit`	`cover` (default), `contain`, `fill`, `none`, `scale-down`	`object-fit` of the image.
`radius`	`xs`–`xl` or any CSS value	Rounded corners.
`fallbackSrc`	image URL	Image shown if `src` fails to load.
`loading`	`lazy`, `eager`	Native `<img>` loading hint — `lazy` defers offscreen images.
`srcSet`	string	Responsive sources — passed straight to the underlying `<img>`.
`sizes`	string	Responsive sizes hint — passed straight to the underlying `<img>`.
`bd`	CSS border, e.g. `1px solid gray`	Border.
`bg`	any CSS color	Background color (shows through a transparent image).
`opacity`	number `0`–`1`, e.g. `0.8`	Image opacity.
`m` `mt` `mb` `ml` `mr` `mx` `my` / `p` `pt` … `py`	Mantine spacing token or any CSS value	Margin / padding spacing props.
`w` `h` `miw` `mih` `maw` `mah`	Mantine size token or any CSS value	Width / height sizing props.

CSS Selectors

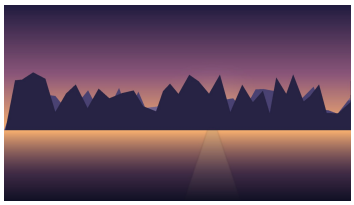
Target the rendered element through its island marker — `[data-aardvark-island="ZoomImage"]` — or through the Mantine Styles API classes (`.mantine-Image-root` and its inner parts):

```
/* Every rendered ZoomImage carries this island marker */
[data-aardvark-island="ZoomImage"] { }

/* Mantine Styles API class on the root element */
.mantine-Image-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:



Source: Markdown

```
{% image src='/landscape.jpg' alt='Mountains at dusk over a still lake' size='35%'
align='left' attr={'onclick': '''
const value = this.alt;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'image', src='/landscape.jpg', alt='Mountains at dusk over a still
lake',
          size='35%', align='left', attr={'onclick': '''
const value = this.alt;
console.log('attr demo value:', value);
alert(value);
'''})
```

Indicator

A small dot or label badge positioned over the corner of another element — a notification count on an avatar, an online status on an icon, and so on. The block body is the element being marked; the indicator is drawn over it. Set `label` for a count, or omit it for a plain dot.

Use it as `{% indicator %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'indicator', …)`. Close the tag with `{% endIndicator %}`.

Demonstrations

Label, dot, and a pulsing status

Set `label` for a count; omit it for a plain dot; add `processing=true` for a pulsing animation (a live status):

3

Source: Markdown

```
{% indicator label='3' color='red' %}
{% avatar name='Ada Lovelace' %}
{% endIndicator %}

{% indicator color='blue' %}
{% avatar name='Alan Turing' %}
{% endIndicator %}

{% indicator color='green' processing=true %}
{% avatar name='Grace Hopper' %}
{% endIndicator %}
```

Source: Python

```
component('aardvark', 'indicator', label='3', color='red',
          children=component('aardvark', 'avatar', name='Ada Lovelace'))
component('aardvark', 'indicator', color='blue',
          children=component('aardvark', 'avatar', name='Alan Turing'))
component('aardvark', 'indicator', color='green', processing=True,
          children=component('aardvark', 'avatar', name='Grace Hopper'))
```

Position, border, and offset

`position` places the indicator at any corner or edge — `top-start`, `top-center`, `top-end` (the default), `middle-start`, …, `bottom-end`. `withBorder` rings it; `offset` nudges it inward (useful over a rounded target); `radius` rounds a labelled badge.

1

2

9

Source: Markdown

```
{% indicator label='1' position='top-start' %}
{% avatar name='A' %}
{% endIndicator %}

{% indicator label='2' position='bottom-end' withBorder=true %}
{% avatar name='B' %}
{% endIndicator %}

{% indicator label='9' position='top-end' offset=4 radius='sm' %}
{% avatar name='C' %}
{% endIndicator %}
```

Source: Python

```
component('aardvark', 'indicator', label='1', position='top-start',
          children=component('aardvark', 'avatar', name='A'))
component('aardvark', 'indicator', label='2', position='bottom-end', withBorder=True,
          children=component('aardvark', 'avatar', name='B'))
component('aardvark', 'indicator', label='9', position='top-end', offset=4, radius='sm',
          children=component('aardvark', 'avatar', name='C'))
```

Size, disabled, and autoContrast

`size` sets the dot diameter in pixels. `disabled` renders only the child (no indicator) — handy for toggling the badge in a loop. `autoContrast` picks a readable label color for the background.

5

2

0

Source: Markdown

```
{% indicator color='grape' size=16 label='5' %}
{% avatar name='D' %}
{% endIndicator %}

{% indicator color='yellow' label='2' autoContrast=true %}
{% avatar name='E' %}
{% endIndicator %}

{% indicator label='0' disabled=true %}
{% avatar name='F' %}
{% endIndicator %}
```

Source: Python

```
component('aardvark', 'indicator', color='grape', size=16, label='5',
          children=component('aardvark', 'avatar', name='D'))
component('aardvark', 'indicator', color='yellow', label='2', autoContrast=True,
          children=component('aardvark', 'avatar', name='E'))
component('aardvark', 'indicator', label='0', disabled=True,
          children=component('aardvark', 'avatar', name='F'))
```

With other components

The body can be any element. Here a status dot rides on an `{% icon %}`; `inline` keeps the wrapper flowing inline with surrounding text:

Source: Markdown

```
{% indicator color='green' size=10 offset=4 inline=true %}
{% icon 'bell' size='lg' %}
{% endIndicator %}
```

Source: Python

```
component('aardvark', 'indicator', color='green', size=10, offset=4, inline=True,
          children=component('aardvark', 'icon', 'bell', size='lg'))
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`label`	string	Text/count shown in the indicator. Omit it for a plain dot.
`color`	any theme color	The indicator's color.
`size`	integer (pixels; `10` by default)	Dot diameter.
`position`	`top-start`, `top-center`, `top-end` (default), `middle-start`, `middle-center`, `middle-end`, `bottom-start`, `bottom-center`, `bottom-end`	Where the indicator sits over the child.

`offset`	integer (pixels)	Nudge inward — useful over a rounded target.
`radius`	`xs`, `sm`, `md`, `lg`, `xl`, or any CSS value	Corner rounding of a labelled badge.
`withBorder`	bool flag (default `false`)	Draw a border around the indicator.
`processing`	bool flag (default `false`)	A pulsing animation.
`disabled`	bool flag (default `false`)	Render only the child (no indicator).
`inline`	bool flag (default `false`)	Lay the container out as an inline element.
`autoContrast`	bool flag (default `false`)	Auto-pick a readable label color for the background.
body	Markdown / components	The element the indicator is positioned over.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Indicator"]` — or through the Mantine Styles API classes (`.mantine-Indicator-root` and its inner parts):

```
/* Every rendered Indicator carries this island marker */
[data-aardvark-island="Indicator"] { }

/* Mantine Styles API class on the root element */
.mantine-Indicator-root { }
.mantine-Indicator-indicator { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

3

Source: Markdown

```
{% indicator label='3' color='red' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}{% avatar name='Ada Lovelace' %}{% endIndicator %}
```

Source: Python

```
component('aardvark', 'indicator', label='3', color='red', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''}, children=component('aardvark', 'avatar', name='Ada Lovelace'))
```

Kbd

Renders a `<kbd>` styled like a physical keyboard key — for documenting shortcuts such as Ctrl, ⌘, or Enter. The key text is the block body, or a `text` attribute when you don't need a body.

Use it as `{% kbd %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'kbd', …)`. Close the tag with `{% endKbd %}`.

Demonstrations

A shortcut, inline

The block body is the key text. Drop keys inline in a sentence:

Press

Ctrl

+

C

to copy.

Source: Markdown

```
Press {% kbd %}Ctrl{% endKbd %} + {% kbd %}C{% endKbd %} to copy.
```

Source: Python

```
ctrl = component('aardvark', 'kbd', children='Ctrl')
c = component('aardvark', 'kbd', children='C')
f"Press {ctrl} + {c} to copy."
```

Sizes

`size` scales the key from `xs` to `xl` (`sm` is the default):

esc

tab

shift

enter

space

Source: Markdown

```
{% kbd size='xs' %}esc{% endKbd %}
{% kbd size='sm' %}tab{% endKbd %}
{% kbd size='md' %}shift{% endKbd %}
{% kbd size='lg' %}enter{% endKbd %}
{% kbd size='xl' %}space{% endKbd %}
```

Source: Python

```
component('aardvark', 'kbd', size='xs', children='esc')
component('aardvark', 'kbd', size='sm', children='tab')
component('aardvark', 'kbd', size='md', children='shift')
component('aardvark', 'kbd', size='lg', children='enter')
component('aardvark', 'kbd', size='xl', children='space')
```

The `text` shorthand

When you don't need a block body, set `text` instead — handy in a loop or a one-liner:

⌘

K

Source: Markdown

```
{% kbd text='⌘' %} {% kbd text='K' %}
```

Source: Python

```
component('aardvark', 'kbd', text='⌘')
component('aardvark', 'kbd', text='K')
```

With other components

Keys read naturally inside running `{% text %}`, where they document a shortcut mid-sentence:

Open the command palette with

⌘

K

.

Source: Markdown

```
{% text %}Open the command palette with {% kbd text='⌘' %} {% kbd text='K' %}.{% endText %}
```

Source: Python

```
cmd = component('aardvark', 'kbd', text='⌘')
k = component('aardvark', 'kbd', text='K')
component('aardvark', 'text', children=f'Open the command palette with {cmd} {k}.')
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`text`	string	The key text, when not using the block body.
`size`	`xs`, `sm` (default), `md`, `lg`, `xl`	Scales the key.
body	text	The key text (the rich form; takes precedence over `text`).

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Kbd"]` — or through the Mantine Styles API classes (`.mantine-Kbd-root` and its inner parts):

```
/* Every rendered Kbd carries this island marker */
[data-aardvark-island="Kbd"] { }

/* Mantine Styles API class on the root element */
.mantine-Kbd-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Ctrl

Source: Markdown

```
{% kbd attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Ctrl{% endKbd %}
```

Source: Python

```
component('aardvark', 'kbd', children='Ctrl', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

LineChart

A **line chart** for trends over time or any ordered axis. Give it the rows in `data` (a JSON array of objects) and the lines in `series` (a JSON array of `{name, color}`), and name the x-axis field with `dataKey`. It renders in the browser (it's Recharts-backed) and hydrates into an interactive island.

Use it as `{% linechart %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'linechart', …)`.

A basic line chart

Each `series` entry names a key in the row objects and the color to draw it; `dataKey` is the x-axis field.

Source: Markdown

```
{% linechart h=260 dataKey='month' withLegend=true data='[
  {"month":"Jan","Signups":120,"Active":80},
  {"month":"Feb","Signups":180,"Active":110},
  {"month":"Mar","Signups":160,"Active":135},
  {"month":"Apr","Signups":240,"Active":190}
]' series='[{"name":"Signups","color":"indigo.6"},{"name":"Active","color":"teal.6"}]' %}
```

Source: Python

```
component('aardvark', 'linechart', h=260, dataKey='month', withLegend=True,
          data='[{"month":"Jan","Signups":120,"Active":80}, …]',
          series='[{"name":"Signups","color":"indigo.6"},{"name":"Active","color":"teal.6"}]')
```

Attributes

Attribute	Valid values	Description
`data`	JSON array of row objects	The chart rows.
`series`	JSON array of `{name, color}`	One line per entry; `name` is a key in each row, `color` a Mantine color.
`dataKey`	string	The row field used for the x-axis.
`h`	integer (px, default `300`)	Chart height.
`curveType`	`monotone`, `linear`, `natural`, `step`, …	Line interpolation.
`unit`	string	Unit appended to axis/tooltip values.
`withLegend`	bool (`true` / `false`)	Show the legend.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="LineChart"]` (or the more specific `[data-aardvark-lib="charts"][data-aardvark-island="LineChart"]` when several libraries share the page) — or through the Mantine Styles API classes (`.mantine-LineChart-root` and its inner parts):

```
/* Every rendered LineChart carries this island marker */
[data-aardvark-island="LineChart"] { }

/* Scope by library + island when you have several libraries in play */
[data-aardvark-lib="charts"][data-aardvark-island="LineChart"] { }

/* Mantine Styles API class on the root element */
.mantine-LineChart-root { }
.mantine-LineChart-container { }
.mantine-LineChart-axis { }
.mantine-LineChart-line { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% linechart h=260 dataKey='month' withLegend=true data='[
  {"month":"Jan","Signups":120,"Active":80},
  {"month":"Feb","Signups":180,"Active":110}
]' series='[{"name":"Signups","color":"indigo.6"},{"name":"Active","color":"teal.6"}]'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'linechart', h=260, dataKey='month', withLegend=True,
          data='[{"month":"Jan","Signups":120,"Active":80},{"month":"Feb","Signups":180,"Active":110}]
',
          series='[{"name":"Signups","color":"indigo.6"},{"name":"Active","color":"teal.6"}]',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

NumberFormatter

A built-in tag that formats a number for display — thousands separators, a fixed number of decimals, a custom decimal separator, and a currency-style prefix or suffix. It renders inline text, so it drops straight into a sentence. Give it a `value` plus any of the formatting options; omit an option to take its default.

Use it as `{% numberformatter value='1234' %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'numberformatter', …)`.

Currency with thousands grouping

A `prefix`, comma grouping (the bare `thousandSeparator` flag), and two fixed decimal places.

Source: Markdown

```
{% numberformatter value='1234567.89' prefix='$' thousandSeparator decimalScale=2
fixedDecimalScale=true %}
```

Source: Python

```
component('aardvark', 'numberformatter', value='1234567.89', prefix='$',
          thousandSeparator=True, decimalScale=2, fixedDecimalScale=True)
```

Plain thousands grouping

The bare `thousandSeparator` flag groups with commas and nothing else.

Source: Markdown

```
{% numberformatter value='9999' thousandSeparator %}
```

Source: Python

```
component('aardvark', 'numberformatter', value='9999', thousandSeparator=True)
```

A suffix

`suffix` appends text after the number — here a percent sign.

Source: Markdown

```
{% numberformatter value='98.6' suffix='%' %}
```

Source: Python

```
component('aardvark', 'numberformatter', value='98.6', suffix='%')
```

A custom separator (European style)

Pass a string to `thousandSeparator` for a custom group separator and set `decimalSeparator` for the decimal point — here a space groups and a comma is the decimal point, padded to two places.

Source: Markdown

```
{% numberformatter value='1234567.5' thousandSeparator=' ' decimalSeparator=','
decimalScale=2 fixedDecimalScale=true %}
```

Source: Python

```
component('aardvark', 'numberformatter', value='1234567.5', thousandSeparator=' ',
          decimalSeparator=',', decimalScale=2, fixedDecimalScale=True)
```

With other components

Because it renders inline text, a formatted number drops into any other component's body — here inside a card subtitle composed in Python.

↗Total revenue

this quarter.

Source: Markdown

```
{% card title="Total revenue" subtitle="Up and to the right." icon="trending-up" %}
{% numberformatter value='2480000' prefix='$' thousandSeparator %} this quarter.
{% endCard %}
```

Source: Python

```
amount = component('aardvark', 'numberformatter', value='2480000', prefix='$',
thousandSeparator=True)
component('aardvark', 'card', title='Total revenue', subtitle='Up and to the right.',
          icon='trending-up', children=amount + ' this quarter.')
```

Attributes

Attribute	Valid values	Description
`value`	string or number	The number to format. Rides as a string and is parsed (`"1234.5"` works).
`prefix`	string	Text shown before the number, e.g. `$`.
`suffix`	string	Text shown after the number, e.g. `USD` or `%`.
`thousandSeparator`	bare flag, or a separator string	A bare flag groups with commas; pass a string (e.g. `' '` or `'.'`) for a custom separator.
`decimalScale`	int	Maximum number of decimal places. A deliberate `0` means no decimals.
`decimalSeparator`	string	The character used for the decimal point.
`fixedDecimalScale`	bool (default `false`)	Pad to `decimalScale` with trailing zeros.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="NumberFormatter"]` — it renders a plain `<span>` carrying the formatted value:

```
/* Every rendered NumberFormatter carries this island marker */
[data-aardvark-island="NumberFormatter"] { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% numberformatter value='1234567.89' prefix='$' thousandSeparator=',' decimalScale=2
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
```

```
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'numberformatter', value='1234567.89', prefix='$',
          thousandSeparator=',', decimalScale=2, attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

OverflowList

A built-in tag for a single-line row of labels (rendered as badges) that collapses any overflow into a trailing “+N” counter when they don’t all fit. It measures the items at runtime and re-fits as the row resizes — narrow the window and watch the counter grow. Hover the counter to see the hidden labels. Pass the labels in `items`, either comma-separated or as a JSON array.

Use it as `{% overflowlist items='A, B, C' %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'overflowlist', …)`.

A row of labels

The default look: light badges, comma-separated `items`. Try narrowing the window — the trailing counter grows to absorb whatever no longer fits.

Source: Markdown

```
{% overflowlist items='React, Vue, Svelte, Solid, Angular, Preact, Qwik, Lit, Ember' %}
```

Source: Python

```
component('aardvark', 'overflowlist',
          items='React, Vue, Svelte, Solid, Angular, Preact, Qwik, Lit, Ember')
```

Items as a JSON array

`items` also accepts a JSON array of strings — handy when a label itself contains a comma.

Source: Markdown

```
{% overflowlist items='["Engineering", "Design, Research", "Sales & Marketing",
"Operations"]' %}
```

Source: Python

```
component('aardvark', 'overflowlist',
          items='["Engineering", "Design, Research", "Sales & Marketing", "Operations"]')
```

Badge variant, color, and size

The badges take the usual Mantine `variant`, `color`, and `size` — here filled indigo badges at the large size.

Source: Markdown

```
{% overflowlist variant='filled' color='indigo' size='lg' items='design, build, ship,
measure, iterate, repeat' %}
```

Source: Python

```
component('aardvark', 'overflowlist', variant='filled', color='indigo', size='lg',
          items='design, build, ship, measure, iterate, repeat')
```

Wider spacing

`gap` sets the space between badges — a Mantine size token or a px value.

Source: Markdown

```
{% overflowlist gap='md' variant='outline' items='alpha, beta, gamma, delta, epsilon, zeta,
eta' %}
```

Source: Python

```
component('aardvark', 'overflowlist', gap='md', variant='outline',
          items='alpha, beta, gamma, delta, epsilon, zeta, eta')
```

Without the counter tooltip

The hidden labels show in a tooltip on the counter by default. Set `withTooltip=false` to turn it off — the counter still collapses overflow, just without the hover reveal.

Source: Markdown

```
{% overflowlist withTooltip=false items='one, two, three, four, five, six, seven, eight' %}
```

Source: Python

```
component('aardvark', 'overflowlist', withTooltip=False,
          items='one, two, three, four, five, six, seven, eight')
```

With other components

Drop a row of tags into any other component's body — here inside a card composed in Python.

Stack

Source: Markdown

```
{% card title="Stack" subtitle="What this service is built on." icon="stack-2" %}
{% overflowlist variant='light' color='grape' items='TypeScript, React, Postgres, Redis,
Docker, Kubernetes, Terraform' %}
{% endCard %}
```

Source: Python

```
tags = component('aardvark', 'overflowlist', variant='light', color='grape',
                 items='TypeScript, React, Postgres, Redis, Docker, Kubernetes, Terraform')
component('aardvark', 'card', title='Stack', subtitle='What this service is built on.',
          icon='stack-2', children=tags)
```

Progressive enhancement

Measurement happens in the browser, so before JavaScript runs (and in the build-time prerender) every item shows with no counter — the collapse is pure progressive enhancement. The first item always stays visible even in a very narrow container.

Attributes

Attribute	Valid values	Description
`items`	comma-separated string, or a JSON array of strings	The labels. A comma-separated string splits on commas; a JSON array (`'["a","b"]'`) keeps commas inside a label.
`variant`	`light` (default), `filled`, `outline`, `dot`, `transparent`, `white`, `default`, `gradient`	Badge variant.
`color`	a Mantine color name or hex	Badge color.
`size`	`xs`, `sm`, `md` (default), `lg`, `xl`	Badge size.
`gap`	a Mantine size token (`xs`–`xl`) or a px value	Spacing between badges (default `xs`).
`withTooltip`	bool (default `true`)	Show the hidden labels in a tooltip on the counter. Set `false` to disable.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="OverflowList"]` — or through the Mantine Styles API classes (`.mantine-OverflowList-root` and its inner parts):

```
/* Every rendered OverflowList carries this island marker */
[data-aardvark-island="OverflowList"] { }

/* Mantine Styles API class on the root element */
.mantine-OverflowList-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% overflowlist items='React, Vue, Svelte, Solid, Angular, Preact, Qwik, Lit, Ember'
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'overflowlist',
          items='React, Vue, Svelte, Solid, Angular, Preact, Qwik, Lit, Ember',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

RollingNumber

A built-in tag for an animated counter — the digits **roll to `value`, re-animating each time the value changes**, the way a live dashboard figure ticks as it updates. It renders inline, so it drops into a sentence or heading, and the usual `size`, `color`, and `fw` props style it. Give it a `value` to show; the roll runs over 600 ms by default. On a static page it shows the final value (there's no value change to animate).

Use it as `{% rollingnumber value='1000' %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'rollingnumber', …)`.

A styled stat

A large grape-colored count, grouped with commas by default.

Source: Markdown

```
{% rollingnumber value='128000' size='2.5rem' fw='700' color='grape' %}
```

Source: Python

```
component('aardvark', 'rollingnumber', value='128000', size='2.5rem', fw='700',
color='grape')
```

A faster, linear roll

Shorten `animationDuration` (in milliseconds) and set `timingFunction='linear'` for a steady, quick count.

Source: Markdown

```
{% rollingnumber value='99' animationDuration=800 timingFunction='linear' size='xl' fw='700'
%}
```

Source: Python

```
component('aardvark', 'rollingnumber', value='99', animationDuration=800,
          timingFunction='linear', size='xl', fw='700')
```

A decimal stat

`decimalScale` fixes the number of decimal places — here a `4.9` rating shown to one decimal — and `animationDuration` lengthens the roll.

Source: Markdown

```
{% rollingnumber value='4.9' decimalScale=1 animationDuration=2000 size='xl' fw='700'
color='teal' %}
```

Source: Python

```
component('aardvark', 'rollingnumber', value='4.9', decimalScale=1,
          animationDuration=2000, size='xl', fw='700', color='teal')
```

Without comma grouping

Comma grouping is on by default. Set `thousandSeparator=false` to count a bare number — useful for years, IDs, or counts that shouldn't read as a quantity.

Source: Markdown

```
{% rollingnumber value='2026' thousandSeparator=false size='xl' fw='700' color='blue' %}
```

Source: Python

```
component('aardvark', 'rollingnumber', value='2026', thousandSeparator=False,
          size='xl', fw='700', color='blue')
```

Prefix and suffix

`prefix` and `suffix` wrap the number with fixed text — a currency symbol, a percent sign, a unit — so the whole figure rolls as one.

Source: Markdown

```
{% rollingnumber value='1299' prefix='$' size='xl' fw='700' color='green' %}

{% rollingnumber value='98' suffix='%' size='xl' fw='700' color='blue' %}
```

Source: Python

```
component('aardvark', 'rollingnumber', value='1299', prefix='$', size='xl', fw='700',
color='green')

component('aardvark', 'rollingnumber', value='98', suffix='%', size='xl', fw='700',
color='blue')
```

With other components

Because it renders inline, a rolling stat drops into any other component's body — here as the headline figure in a card composed in Python.

Active installs

54213 developers

Source: Markdown

```
{% card title="Active installs" subtitle="And climbing." icon="download" %}
{% rollingnumber value='54213' size='2rem' fw='700' color='indigo' %} developers
{% endCard %}
```

Source: Python

```
stat = component('aardvark', 'rollingnumber', value='54213', size='2rem', fw='700',
color='indigo')
component('aardvark', 'card', title='Active installs', subtitle='And climbing.',
          icon='download', children=stat + ' developers')
```

Attributes

Attribute	Valid values	Description
`value`	string or number	The number to show. Rides as a string and is parsed (`"1234"` and `"12.5"` both work). The roll animates whenever this value changes.
`animationDuration`	int (milliseconds)	Roll length (default `600`). A deliberate `0` jumps straight to the value.
`timingFunction`	`linear`, `ease` (default), `ease-in`, `ease-out`, `ease-in-out`, or any CSS timing function	The roll's easing curve.
`decimalScale`	int	Fixed number of decimal places (defaults to the value's own).
`fixedDecimalScale`	bool (default `false`)	Pad with trailing zeros to match `decimalScale`.
`thousandSeparator`	bool (default `true`)	Group with commas. Set `false` to disable.
`prefix`	string	Fixed text before the number (e.g. `$`).
`suffix`	string	Fixed text after the number (e.g. `%`).
`size`	a Mantine size token (`xs`–`xl`) or a CSS size (e.g. `2.5rem`)	Font size.
`color`	a Mantine color name or hex	Text color.
`fw`	a CSS font-weight (e.g. `700`)	Font weight.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="RollingNumber"]` — or through the Mantine Styles API classes (`.mantine-RollingNumber-root` and its inner parts):

```
/* Every rendered RollingNumber carries this island marker */
```

```
[data-aardvark-island="RollingNumber"] { }

/* Mantine Styles API class on the root element */
.mantine-RollingNumber-root { }
.mantine-RollingNumber-digit { }
.mantine-RollingNumber-char { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% rollingnumber value='128000' size='2.5rem' fw='700' color='grape' attr={'onclick': '''
const value = this.getAttribute('aria-label');
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'rollingnumber', value='128000', size='2.5rem', fw='700',
color='grape',
          attr={'onclick': '''
const value = this.getAttribute('aria-label');
console.log('attr demo value:', value);
alert(value);
'''})
```

Spoiler

A **built-in** tag that hides long content behind a **show-more** toggle. When the block body is taller than `maxHeight`, a toggle appears and the reader expands the rest on demand; the body renders as ordinary Markdown. Use it as `{% spoiler %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'spoiler', …)`.

Demonstrations

A bare spoiler collapses its body once the content is taller than `maxHeight` (60px here), and the **Show more** toggle reveals the rest:

This region starts collapsed once its content is taller than `maxHeight`. A **Show more** toggle reveals the rest, and **Hide** collapses it again. The body is ordinary Markdown — **bold**, `code`, links, and lists all work.

- It is good for long change notes, FAQs, and footnote-y asides.
- The toggle only appears when the content actually overflows `maxHeight`.

Source: Markdown

```
{% spoiler maxHeight=60 %}
This region starts collapsed once its content is taller than `maxHeight`.
A **Show more** toggle reveals the rest, and **Hide** collapses it again.
The body is ordinary Markdown — **bold**, `code`, [links](/), and lists all work.

- It is good for long change notes, FAQs, and footnote-y asides.
- The toggle only appears when the content actually overflows `maxHeight`.
{% endSpoiler %}
```

Source: Python

```
component('aardvark', 'spoiler', maxHeight=60, children=(
    "This region starts collapsed once its content is taller than `maxHeight`.\n"
    "A **Show more** toggle reveals the rest, and **Hide** collapses it again.\n\n"
    "- It is good for long change notes, FAQs, and footnote-y asides.\n"
    "- The toggle only appears when the content actually overflows `maxHeight`.\n"
))
```

Custom labels

`showLabel` and `hideLabel` set the toggle text for the collapsed and expanded states:

The toggle text is yours to set. Keep `showLabel` short — it sits inline under the collapsed content. This body is long enough to overflow the 40px `maxHeight`, so the toggle shows.

Source: Markdown

```
{% spoiler maxHeight=40 showLabel='Read the details' hideLabel='Collapse' %}
The toggle text is yours to set. Keep `showLabel` short — it sits inline under the
collapsed content. This body is long enough to overflow the 40px `maxHeight`, so the
toggle shows.
{% endSpoiler %}
```

Source: Python

```
component('aardvark', 'spoiler', maxHeight=40,
          showLabel='Read the details', hideLabel='Collapse',
          children='The toggle text is yours to set…')
```

Instant reveal

`transitionDuration=0` disables the reveal animation, so the body appears immediately:

With `transitionDuration=0` the content snaps open and shut with no slide. Useful when you want the toggle behavior without any motion. This paragraph is tall enough to overflow the 40px `maxHeight`, so the toggle appears.

Source: Markdown

```
{% spoiler maxHeight=40 transitionDuration=0 %}
With `transitionDuration=0` the content snaps open and shut with no slide. Useful when you
want the toggle behavior without any motion. This paragraph is tall enough to overflow the
40px `maxHeight`, so the toggle appears.
{% endSpoiler %}
```

Source: Python

```
component('aardvark', 'spoiler', maxHeight=40, transitionDuration=0,
          children='With transitionDuration=0 the content snaps open…')
```

With other components

A spoiler body is full Markdown, so it can hold any other component — here a {% card %} and a {% badge %} stay hidden until the reader expands the region:

Recent updates [New]

v2.0

A major release with breaking changes and a migration guide.

Source: Markdown

```
{% spoiler maxHeight=50 showLabel='Show the changelog' hideLabel='Hide' %}
Recent updates {% badge color='green' %}New{% endBadge %}

{% card title="v2.0" %}
A major release with breaking changes and a migration guide.
{% endCard %}
{% endSpoiler %}
```

Source: Python

```
body = "Recent updates " + component('aardvark', 'badge', color='green', children='New')
body += "\n\n" + component('aardvark', 'card', title='v2.0',
                           children='A major release with breaking changes and a migration
guide.')
page.print(component('aardvark', 'spoiler', maxHeight=50,
                     showLabel='Show the changelog', hideLabel='Hide', children=body))
```

Attributes

Attribute	Valid values	Description
(body)	Markdown	The collapsible content. Rendered as Markdown. Close the tag with `{% endSpoiler %}`.
`maxHeight`	integer (pixels)	Visible height before the toggle appears. Defaults to `100`. The toggle only shows when the body is taller than this.
`showLabel`	string	Toggle text while collapsed. Defaults to `Show more`.
`hideLabel`	string	Toggle text while expanded. Defaults to `Hide`.
`transitionDuration`	integer (ms)	Reveal animation duration. Defaults to `200`; set `0` to disable the animation.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Spoiler"]` — or through the Mantine Styles API classes (`.mantine-Spoiler-root` and its inner parts):

```
/* Every rendered Spoiler carries this island marker */
[data-aardvark-island="Spoiler"] { }

/* Mantine Styles API class on the root element */
.mantine-Spoiler-root { }
```

```
.mantine-Spoiler-content { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

A paragraph long enough to be collapsed behind the Show more toggle, then revealed on click.

Source: Markdown

```
{% spoiler maxHeight=60 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
A paragraph long enough to be collapsed behind the Show more toggle, then revealed on click.
{% endSpoiler %}
```

Source: Python

```
component('aardvark', 'spoiler', maxHeight=60,
          children='A paragraph long enough to be collapsed behind the Show more toggle,
then revealed on click.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Timeline

A built-in tag for a vertical timeline — a run of bullets joined by a connecting line, each carrying a title and a short body. Pass the events as a JSON array in `items`; each object takes a `title`, its `content`, and optionally a `bullet` (a Tabler icon name, an emoji, or short text), plus a per-item `color` and `lineVariant`. Set `active` to mark how many items are completed.

Use it as `{% timeline items='[…]' %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'timeline', …)`.

A basic timeline

Three events with Tabler-icon bullets; `active=1` highlights the first two as completed.

Source: Markdown

```
{% timeline active=1 items='[
  {"title": "Cloned", "content": "Repository created and pushed.", "bullet": "git-branch"},
  {"title": "Built", "content": "First successful vark build.", "bullet": "package"},
  {"title": "Deployed", "content": "Live on the CDN.", "bullet": "rocket"}
]' %}
```

Source: Python

```
component('aardvark', 'timeline', active=1, items='''[
  {"title": "Cloned", "content": "Repository created and pushed.", "bullet": "git-branch"},
  {"title": "Built", "content": "First successful vark build.", "bullet": "package"},
  {"title": "Deployed", "content": "Live on the CDN.", "bullet": "rocket"}
]''')
```

A bullet that looks like a bare Tabler icon name renders that glyph; an emoji, number, or short text renders as-is; omit it for Mantine’s default dot.

Highlighting progress with a dashed segment

`active=0` marks the first item completed. A per-item `lineVariant` of `dashed` shows the next segment as in-progress, and `color='teal'` accents the active bullets and line.

Source: Markdown

```
{% timeline active=0 color='teal' items='[
  {"title": "Shipped", "content": "Done.", "bullet": "check"},
  {"title": "In review", "content": "Almost there.", "bullet": "clock", "lineVariant":
"dashed"},
  {"title": "Planned", "content": "Not started.", "bullet": "circle"}
]' %}
```

Source: Python

```
component('aardvark', 'timeline', active=0, color='teal', items='''[
  {"title": "Shipped", "content": "Done.", "bullet": "check"},
  {"title": "In review", "content": "Almost there.", "bullet": "clock", "lineVariant":
"dashed"},
  {"title": "Planned", "content": "Not started.", "bullet": "circle"}
]''')
```

Sizing, alignment, and reverse highlighting

`bulletSize` and `lineWidth` are in px; `radius` rounds the bullets; `align='right'` runs the line down the right; and `reverseActive` highlights from the bottom up.

Source: Markdown

```
{% timeline active=1 align='right' bulletSize=28 lineWidth=3 radius='sm' reverseActive
color='grape' items='[
  {"title": "Now", "content": "Latest release.", "bullet": "⎚"},
  {"title": "Last month", "content": "Previous release.", "bullet": "⎚"},
  {"title": "Earlier", "content": "First release.", "bullet": "⎚"}
]' %}
```

Source: Python

```
component('aardvark', 'timeline', active=1, align='right', bulletSize=28, lineWidth=3,
          radius='sm', reverseActive=True, color='grape', items='''[
  {"title": "Now", "content": "Latest release.", "bullet": "⎚"},
  {"title": "Last month", "content": "Previous release.", "bullet": "⎚"},
  {"title": "Earlier", "content": "First release.", "bullet": "⎚"}
]''')
```

Per-item color and autoContrast

Each item can override the accent `color`; `autoContrast` picks a readable icon color against the bullet fill.

Source: Markdown

```
{% timeline active=2 autoContrast items='[
  {"title": "Triage", "content": "Issue opened.", "bullet": "flag", "color": "blue"},
  {"title": "Fix", "content": "Patch merged.", "bullet": "git-merge", "color": "green"},
  {"title": "Release", "content": "Shipped to users.", "bullet": "rocket", "color": "grape"}
]' %}
```

Source: Python

```
component('aardvark', 'timeline', active=2, autoContrast=True, items='''[
  {"title": "Triage", "content": "Issue opened.", "bullet": "flag", "color": "blue"},
  {"title": "Fix", "content": "Patch merged.", "bullet": "git-merge", "color": "green"},
  {"title": "Release", "content": "Shipped to users.", "bullet": "rocket", "color": "grape"}
]''')
```

With other components

A timeline drops into any other component's body — here inside a card, built in Python from a list so each event can come from data.

Release history

Source: Markdown

```
{% card title="Release history" subtitle="The last three milestones." icon="history" %}
{% timeline active=2 items='[
  {"title": "v0.1.0", "content": "First public build.", "bullet": "tag"},
  {"title": "v0.1.5", "content": "Search and themes.", "bullet": "tag"},
  {"title": "v0.2.0", "content": "Built-in components.", "bullet": "tag"}
]' %}
{% endCard %}
```

Source: Python

```
import json
releases = [
    {'title': 'v0.1.0', 'content': 'First public build.', 'bullet': 'tag'},
    {'title': 'v0.1.5', 'content': 'Search and themes.', 'bullet': 'tag'},
    {'title': 'v0.2.0', 'content': 'Built-in components.', 'bullet': 'tag'},
]
tl = component('aardvark', 'timeline', active=2, items=json.dumps(releases))
component('aardvark', 'card', title='Release history', subtitle='The last three
milestones.',
          icon='history', children=tl)
```

Attributes

Attribute	Valid values	Description
`items`	JSON array of event objects	The events: each `{title, content, bullet?, color?, lineVariant?}` (see per-item fields below).
`active`	int	Index of the last highlighted (completed) item; earlier ones highlight too.
`bulletSize`	int (px)	Diameter of the bullet circles.
`lineWidth`	int (px)	Thickness of the connecting line.
`radius`	a Mantine size token (`xs`–`xl`) or a px value	Bullet corner radius.
`color`	a Mantine color name or hex	Accent color for active bullets and the line.
`align`	`left` (default) or `right`	Which side the line runs.
`reverseActive`	bool (default `false`)	Highlight from the bottom up instead of the top down.
`autoContrast`	bool (default `false`)	Auto-pick a readable icon color against the bullet fill.

Per-item fields (objects in `items`)

Field	Valid values	Description
`title`	string	The item’s heading.
`content`	string	The item’s body text.
`bullet`	a Tabler icon name, emoji, number, or short label	Shown in the bullet circle. Omit for Mantine’s default dot.
`color`	a Mantine color name or hex	Override the accent color for just this item.
`lineVariant`	`solid` (default), `dashed`, `dotted`	The line below this item.

CSS Selectors

Target the rendered element through its island marker — `[data-aardvark-island="Timeline"]` — or through the Mantine Styles API classes (`.mantine-Timeline-root` and its inner parts):

```
/* Every rendered Timeline carries this island marker */
[data-aardvark-island="Timeline"] { }

/* Mantine Styles API class on the root element */
.mantine-Timeline-root { }
.mantine-Timeline-item { }
.mantine-Timeline-itemBullet { }
.mantine-Timeline-itemBody { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% timeline active=1 items='[
  {"title": "Cloned", "content": "Repository created and pushed.", "bullet": "git-branch"},
  {"title": "Built", "content": "First successful vark build.", "bullet": "package"}
]' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'timeline', active=1, items='''[
  {"title": "Cloned", "content": "Repository created and pushed.", "bullet": "git-branch"},
  {"title": "Built", "content": "First successful vark build.", "bullet": "package"}
]''', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Typography

Built-in tags for **text and document structure** — every Mantine typography component, wrapped as a single Markdown tag with no setup. Plain Markdown still handles ordinary prose; reach for these when you want the styling, color, icon, or layout controls Mantine adds on top.

- Text — style any run of text: weight, size, color, alignment, transform, gradient, and truncation.
- Title — a heading (`<h1>`–`<h6>`) with an explicit level, visual size, and the full typography surface.
- Blockquote — a styled pull-quote with a citation, accent color, radius, and an optional icon.
- Code — inline or block code, shown verbatim, with an optional background tint.
- Mark — a tinted `<mark>` run for drawing the eye to a phrase mid-sentence.
- Highlight — highlight every match of one or more substrings inside a longer run of text.
- List — an ordered or unordered list with sizing, spacing, and an optional icon bullet.
- Table — a data table with borders, striping, hover, a caption, and a sticky header.
- Typography — a prose-styling wrapper that gives a block of raw HTML Mantine's article styles.
- Divider — a labelled or plain horizontal / vertical rule.
- VisuallyHidden — hide content visually while keeping it available to screen readers.

Every tag forwards the underlying Mantine component's options, so anything the component accepts is reachable here. For the full prop reference, follow the Mantine docs link on each page; for anything not surfaced as a tag option, drop down to the raw `component()` island.

Blockquote

`{% blockquote %}` is a **built-in** tag for pull-quotes — a quote set off with an accent bar, an optional citation, and an optional icon. A plain Markdown `>` quote still works for ordinary quotes; reach for this tag when you want the color, citation, or icon. The quote text is the block body (rendered as Markdown) or the plain-text `text` param.

Use it as `{% blockquote %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'blockquote', …)`.

Demonstrations

Basic quote with citation and icon

`cite` adds an attribution line below the quote; `icon` takes a Tabler icon name (fetched lazily at view time) for the corner badge.

Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.

Source: Markdown

```
{% blockquote cite='— Antoine de Saint-Exupéry' icon='quote' %}
Perfection is achieved not when there is nothing more to add, but when there is nothing
left to take away.
{% endBlockquote %}
```

Source: Python

```
component('aardvark', 'blockquote',
          cite='— Antoine de Saint-Exupéry', icon='quote',
          children='Perfection is achieved not when there is nothing more to add, '
                   'but when there is nothing left to take away.')
```

Citation only

Omit `icon` for Mantine's default (no badge). The body renders as Markdown, so emphasis and links format.

That brain of mine is something **more than merely mortal**, as time will show.

Source: Markdown

```
{% blockquote cite='— Ada Lovelace' %}
That brain of mine is something **more than merely mortal**, as time will show.
{% endBlockquote %}
```

Source: Python

```
component('aardvark', 'blockquote', cite='— Ada Lovelace',
          children='That brain of mine is something **more than merely mortal**, '
                   'as time will show.')
```

Color and radius

`color` tints the accent bar and icon (any theme color); `radius` rounds the corners (`xs`–`xl` or any CSS value).

Make it work, make it right, make it fast.

Source: Markdown

```
{% blockquote color='grape' radius='lg' icon='heart' %}
Make it work, make it right, make it fast.
{% endBlockquote %}
```

Source: Python

```
component('aardvark', 'blockquote', color='grape', radius='lg', icon='heart',
          children='Make it work, make it right, make it fast.')
```

Icon size and the text shortcut

`iconSize` adjusts the badge size in pixels. For a one-liner, the `text` param is the plain-text shortcut for the body.

A good idea, well framed.

Source: Markdown

```
{% blockquote color='blue' icon='bulb' iconSize=48 text='A good idea, well framed.' %}{%
endBlockquote %}
```

Source: Python

```
component('aardvark', 'blockquote', color='blue', icon='bulb', iconSize=48,
          text='A good idea, well framed.')
```

With other components

The block body is full Markdown, so it can host other built-in tags — here a Mark highlight and inline Code inside the quote.

Run

vark build

and you get a

static site

.

Source: Markdown

```
{% blockquote color='teal' icon='code' cite='— the docs' %}
Run {% code %}vark build{% endCode %} and you get a {% mark color='lime' %}static site{%
endMark %}.
{% endBlockquote %}
```

Source: Python

```
body = ("Run " + component('aardvark', 'code', children='vark build')
        + " and you get a "
        + component('aardvark', 'mark', color='lime', children='static site') + ".")
component('aardvark', 'blockquote', color='teal', icon='code', cite='— the docs',
          children=body)
```

Attributes

Attribute	Valid values	Description
`text`	Any string	The quote text, used when there's no block body (plain text — not rendered as Markdown).
`cite`	Any string	Attribution line shown below the quote.
`color`	Any theme color (e.g. `blue`, `grape`, `teal`)	Tints the accent bar and the icon.
`radius`	`xs` `sm` `md` `lg` `xl`, or any CSS length	Corner radius.
`icon`	A Tabler icon name (e.g. `quote`, `bulb`, `heart`)	Glyph for the corner badge; fetched lazily at view time.
`iconSize`	Integer (pixels)	Size of the icon badge.

The quote itself is the block body (rendered as Markdown — paragraphs, emphasis, links all work), or the `text` param for a plain one-liner.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Blockquote"]`, or through the Mantine Styles API classes:

```
/* Every rendered Blockquote carries this island marker */
[data-aardvark-island="Blockquote"] { }

/* Mantine Styles API classes */
.mantine-Blockquote-root { }
.mantine-Blockquote-icon { }
.mantine-Blockquote-cite { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.

Source: Markdown

```
{% blockquote cite='— Antoine de Saint-Exupéry' icon='quote' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
Perfection is achieved not when there is nothing more to add, but when there is nothing left
to take away.
{% endBlockquote %}
```

Source: Python

```
component('aardvark', 'blockquote', cite='— Antoine de Saint-Exupéry', icon='quote',
          children='Perfection is achieved not when there is nothing more to add, '
                   'but when there is nothing left to take away.',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Code

`{% code %}` is a **built-in** tag for code styling. It shows its content **verbatim** — no Markdown, no syntax highlighting: by default an inline `<code>` run for a short token mid-sentence, or a standalone block with `block`. For highlighted, multi-line source, use a fenced ``` code block instead; reach for this tag when you want a styled token inline or a quick tinted block.

Use it as `{% code %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'code', …)`.

Demonstrations

Inline (the default)

By default `{% code %}` is **inline** — it sits in a sentence without breaking the paragraph. The code is the block body, or the inline `code` param.

Run

npm install aardvark

to get started.

Source: Markdown

```
Run {% code %}npm install aardvark{% endCode %} to get started.
```

Source: Python

```
component('aardvark', 'code', code='npm install aardvark')
```

Block

Add `block` for a standalone, padded `<pre>`-style block instead of an inline run.

export AARDVARK_KEY=aardvark_live_...

Source: Markdown

```
{% code block=true %}export AARDVARK_KEY=aardvark_live_…{% endCode %}
```

Source: Python

```
component('aardvark', 'code', block=True, children='export AARDVARK_KEY=aardvark_live_…')
```

Color

`color` tints the background — handy for a positive / negative cue.

200 OK

404 Not Found

Source: Markdown

```
{% code color='green' %}200 OK{% endCode %} {% code color='red' %}404 Not Found{% endCode %}
```

Source: Python

```
component('aardvark', 'code', color='green', children='200 OK')
component('aardvark', 'code', color='red', children='404 Not Found')
```

Verbatim content

The content is shown literally — angle brackets and ampersands read as themselves, and Markdown is **not** processed.

<Component prop="value" />

Source: Markdown

```
{% code %}<Component prop="value" />{% endCode %}
```

Source: Python

```
component('aardvark', 'code', code='<Component prop="value" />')
```

With other components

Inline code reads naturally inside other built-ins — here a status token inside a Blockquote, and a tinted token in a List item.

A healthy build ends with

Built 182 page(s)

.

Source: Markdown

```
{% blockquote icon='terminal-2' cite='— the release notes' %}
A healthy build ends with {% code color='green' %}Built 182 page(s){% endCode %}.
{% endBlockquote %}
```

Source: Python

```
body = ("A healthy build ends with "
        + component('aardvark', 'code', color='green', children='Built 182 page(s)') + ".")
component('aardvark', 'blockquote', icon='terminal-2', cite='— the release notes',
          children=body)
```

Attributes

Attribute	Valid values	Description
`code`	Any string	The code text, used when there's no block body. Shown verbatim.
`block`	`true` / `false` (default `false`)	Render a standalone `<pre>`-style block instead of an inline `<code>` run.
`color`	Any theme color (e.g. `green`, `red`)	Background tint.

The code is the block body or the `code` param; either way it's shown literally (no Markdown, no syntax highlighting). For highlighted, multi-line source, use a fenced ``` code block — it gets the site's full syntax highlighting, which `{% code %}` (verbatim by design) does not.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Code"]`, or through the Mantine Styles API classes:

```
/* Every rendered Code carries this island marker */
[data-aardvark-island="Code"] { }

/* Mantine Styles API classes */
.mantine-Code-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Run

npm install aardvark

to get started.

Source: Markdown

```
Run {% code attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}npm install aardvark{% endCode %} to get started.
```

Source: Python

```
component('aardvark', 'code', children='npm install aardvark', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

CodeBlock

`{% CodeBlock %}` is a **runtime** React code block: the source is handed to Shiki in the browser, which tokenizes it and paints the highlighted result, with a copy button on top. It hydrates into an interactive island, so the highlighting is computed live on the page rather than baked in at build time.

This is **separate** from the site's fenced ``` code blocks. Those are highlighted once, at build time, into static HTML — the default for prose. Reach for `{% CodeBlock %}` when you want a code block whose highlighting is produced at runtime by Shiki (for example, to match a Shiki theme exactly, or to drive the source from data), and keep using fenced blocks for ordinary inline documentation.

Use it as `{% CodeBlock %}` in Markdown, or call it from Python logic (loops, snippets) via `component('CodeBlock', …)`.

A basic block

Pass the source as `code` and its `language`. Shiki tokenizes it on mount; until then a plain, un-highlighted block is shown, so the page never waits on the tokenizer.

Source: Markdown

```
{% CodeBlock code='const greeting: string = "hello, world";' language='ts' %}
```

Source: Python

```
component('CodeBlock', code='const greeting: string = "hello, world";', language='ts')
```

Languages and the copy button

`language` accepts any of Shiki's grammar ids (`tsx`, `python`, `bash`, `json`, `css`, …); an unregistered language still renders, just un-tokenized. `withCopyButton` (on by default) toggles the copy control in the top-right corner.

Source: Markdown

```
{% CodeBlock code='print(f"hello, {name}")' language='python' %}

{% CodeBlock code='echo "no copy button here"' language='bash' withCopyButton=false %}
```

Source: Python

```
component('CodeBlock', code='print(f"hello, {name}")', language='python')

component('CodeBlock', code='echo "no copy button here"', language='bash',
withCopyButton=False)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`code`	Any string	The source to highlight. For multi-line source, call it from Python (`component('CodeBlock', code=…)`) where the string can carry real line breaks.
`language`	A Shiki grammar id (`tsx`, `ts`, `python`, `bash`, `json`, …)	Grammar used for tokenizing. Defaults to `tsx`.
`withCopyButton`	bool (`true` / `false`, default `true`)	Show the copy-to-clipboard button.

CSS Selectors

Target the block from your theme CSS through its island marker or Mantine's runtime part classes.

```
/* The island root (the element CodeBlock forwards its ref to) */
[data-aardvark-island="CodeBlock"] {
}

/* Mantine's CodeHighlight parts (the root part is `codeHighlight`) */
.mantine-CodeHighlight-codeHighlight {
}
.mantine-CodeHighlight-pre {
}
.mantine-CodeHighlight-code {
}
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes (including event handlers) straight onto the rendered element.

Source: Markdown

```
{% CodeBlock code='x' language='ts' attr={'onclick': '''
const value = this.innerText;
```

```
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('CodeBlock', code='x', language='ts', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Divider

`divider` is a horizontal or vertical rule — plain, or with a caption. It ships with Aardvark, so a divider is a single tag with no setup. A plain Markdown thematic break (`---`) already renders as a Divider, so reach for the tag when you want a **label**, a **variant**, a specific **color**, or a **vertical** rule.

Use it as `{% divider %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'divider', …)`.

Demonstrations

The label is the block body or a `label` param. A plain `divider` with a centered label, and a dashed one with a body label:

Source: Markdown

```
{% divider label='Section' %}
{% divider variant='dashed' labelPosition='center' %}Part two{% endDivider %}
```

Source: Python

```
component('aardvark', 'divider', label='Section')
component('aardvark', 'divider', variant='dashed', labelPosition='center',
          children='Part two')
```

The label sits in the **center** by default; `labelPosition` moves it `left` or `right`:

Source: Markdown

```
{% divider label='Left' labelPosition='left' %}
{% divider label='Center' %}
{% divider label='Right' labelPosition='right' %}
```

Source: Python

```
for pos in ('left', 'center', 'right'):
    component('aardvark', 'divider', label=pos.title(), labelPosition=pos)
```

`variant`, `color`, and `size` change the line itself — dashed, dotted, or a thick colored rule:

Source: Markdown

```
{% divider variant='dashed' labelPosition='center' %}dashed{% endDivider %}
{% divider variant='dotted' labelPosition='center' %}dotted{% endDivider %}
{% divider color='primary' size='md' labelPosition='center' %}thick & colored{% endDivider
%}
```

Source: Python

```
component('aardvark', 'divider', variant='dashed', labelPosition='center',
          children='dashed')
component('aardvark', 'divider', variant='dotted', labelPosition='center',
          children='dotted')
component('aardvark', 'divider', color='primary', size='md', labelPosition='center',
          children='thick & colored')
```

`my` controls the breathing room around a horizontal rule (a spacing token or any CSS length):

A paragraph above the rule.

A paragraph below the rule — note the wider gap above and below.

Source: Markdown

```
{% divider my='xl' %}
```

Source: Python

```
component('aardvark', 'divider', my='xl')
```

Set `orientation='vertical'` and give the rule an `h` (height); a vertical rule is meant to sit inside a flex row, between two pieces of content:

Home

Docs

About

Source: Markdown

```
<div style="display: flex; align-items: center;">
  <span>Home</span>
  {% divider orientation='vertical' h='1.25rem' mx='sm' %}
  <span>Docs</span>
  {% divider orientation='vertical' h='1.25rem' mx='sm' %}
  <span>About</span>
</div>
```

Source: Python

```
rule = component('aardvark', 'divider', orientation='vertical', h='1.25rem', mx='sm')
'<div style="display: flex; align-items: center;">' \
    '<span>Home</span>' + rule + '<span>Docs</span>' + rule + '<span>About</span></div>'
```

With other components

A divider separates sections inside any surface. Here it splits a `paper` panel:

Account

Deleting your account is permanent.

Source: Markdown

```
{% paper withBorder=true p='lg' radius='md' %}
**Account**
{% divider label='Danger zone' labelPosition='left' color='red' my='md' %}
Deleting your account is permanent.
{% endPaper %}
```

Source: Python

```
inner = (
    '**Account**'
    + component('aardvark', 'divider', label='Danger zone',
                labelPosition='left', color='red', my='md')
    + 'Deleting your account is permanent.'
)
component('aardvark', 'paper', withBorder=True, p='lg', radius='md', children=inner)
```

Attributes

Omit any attribute to take its default. Spacing and sizing values take a Mantine token (`xs`–`xl`) or any CSS length.

Attribute	Valid values	Description
`label`	Plain text (not Markdown)	Caption text, when not using the block body.
`labelPosition`	`center` (default) / `left` / `right`	Where the label sits on the rule.
`orientation`	`horizontal` (default) / `vertical`	Rule direction. A vertical rule needs an `h`.

`variant`	`solid` (default) / `dashed` / `dotted`	Line style.
`size`	`xs`–`xl` (default `xs`)	Line thickness.
`color`	A theme color (`primary`, `blue`, `red`, ...)	Line color. Defaults to the subtle border color.
`m` / `mt` / `mb` / `ml` / `mr` / `mx` / `my`	Spacing token or any CSS value	Margin — `my` sets the space around a horizontal rule.
`p` / `pt` / `pb` / `pl` / `pr` / `px` / `py`	Spacing token or any CSS value	Padding.
`w` / `h`	Any CSS length	Sizing — give a **vertical** rule its `h`.
`miw` / `mih` / `maw` / `mah`	Any CSS length	Min/max width and min/max height.

`attr={...}` forwards raw HTML attributes (e.g. an `id` for a deep link) onto the rendered rule.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Divider"]`, or through the Mantine Styles API classes:

```
/* Every rendered Divider carries this island marker */
[data-aardvark-island="Divider"] { }

/* Mantine Styles API classes */
.mantine-Divider-root { }
.mantine-Divider-label { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% divider label='Section' attr={'onclick': '''
const value = this.innerText;
```

```
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'divider', label='Section', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Highlight

`{% highlight %}` is a **built-in** tag that **scans a run of text and highlights every match** of one or more substrings. Give the term(s) to find and the text to search, and each match is wrapped in a tinted `<mark>`. The text is the block body or a `text` param; it's treated as plain text, so substring matching is reliable.

Use it as `{% highlight %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'highlight', …)`.

Demonstrations

Single term

Give the substring to find in `highlight`; every occurrence in the body is marked.

Write Markdown, get a site. Markdown in, HTML out.

Source: Markdown

```
{% highlight highlight='Markdown' %}Write Markdown, get a site. Markdown in, HTML out.{%
endHighlight %}
```

Source: Python

```
component('aardvark', 'highlight', highlight='Markdown',
          children='Write Markdown, get a site. Markdown in, HTML out.')
```

Multiple terms

Pass several comma-separated substrings in `highlight` and every match of any of them is marked.

Aardvark is fast, simple, and static — a static-site generator.

Source: Markdown

```
{% highlight highlight='fast, simple, static' %}Aardvark is fast, simple, and static — a
static-site generator.{% endHighlight %}
```

Source: Python

```
component('aardvark', 'highlight', highlight=['fast', 'simple', 'static'],
          children='Aardvark is fast, simple, and static — a static-site generator.')
```

Color and text styling

`color` tints the highlight; the Text typography props (`size`, `fw`, `c`, `ta`, `tt`, `td`, `fs`) style the whole run.

Only the matched word is highlighted; the rest is styled text.

Source: Markdown

```
{% highlight highlight='matched' color='lime' fw=600 size='lg' %}Only the matched word is
highlighted; the rest is styled text.{% endHighlight %}
```

Source: Python

```
component('aardvark', 'highlight', highlight='matched', color='lime', fw='600', size='lg',
          children='Only the matched word is highlighted; the rest is styled text.')
```

The text shortcut

For a one-liner, the `text` param is the plain-text alternative to the block body.

Add an id for CSS or JS hooks.

Source: Markdown

```
{% highlight highlight='id' text='Add an id for CSS or JS hooks.' id='hl-demo' color='grape'
%}{% endHighlight %}
```

Source: Python

```
component('aardvark', 'highlight', highlight='id', text='Add an id for CSS or JS hooks.',
          id='hl-demo', color='grape')
```

With other components

Highlight pairs naturally with other typography built-ins — here it marks a term inside a Blockquote.

A static site is fast, a static site is cheap.

Source: Markdown

```
{% blockquote icon='search' color='cyan' %}
{% highlight highlight='static' color='cyan' %}A static site is fast, a static site is
cheap.{% endHighlight %}
{% endBlockquote %}
```

Source: Python

```
inner = component('aardvark', 'highlight', highlight='static', color='cyan',
                  children='A static site is fast, a static site is cheap.')
component('aardvark', 'blockquote', icon='search', color='cyan', children=inner)
```

Attributes

Attribute	Valid values	Description
`text`	Any string	The text to search, used when there's no block body.
`highlight`	One phrase, or several comma-separated (tag); a string or list of strings (Python)	The substring(s) to highlight — every match is wrapped in a `<mark>`.
`color`	Any theme color (e.g. `lime`, `grape`, `cyan`)	Highlight tint.
`size`	`xs` `sm` `md` `lg` `xl`	Text size of the whole run.
`fw`	Font-weight value (e.g. `600`)	Font weight of the run.
`c`	Any theme color	Text color of the run.
`ta`	`left` `center` `right` `justify`	Text align.
`tt`	`uppercase` `lowercase` `capitalize`	Text transform.
`td`	`underline` `line-through` `none`	Text decoration.
`fs`	`italic` `normal`	Font style.
`id`	Any string	HTML `id` on the rendered element, for CSS / JS selectors.

The text to search is the block body or the `text` param. It's plain text — Mantine searches it character by character — so Markdown inside it is **not** formatted. For a single fixed phrase you wrap by hand, Mark is simpler.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Highlight"]`, or through the Mantine Styles API classes. Each matched substring is wrapped in a Mantine `Mark` (`.mantine-Mark-root`). The relevant classes:

```
/* Every rendered Highlight carries this island marker */
[data-aardvark-island="Highlight"] { }

/* Mantine Styles API classes */
.mantine-Highlight-root { }

/* Each highlighted substring */
.mantine-Mark-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Write Markdown, get a site. Markdown in, HTML out.

Source: Markdown

```
{% highlight highlight='Markdown' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Write Markdown, get a site. Markdown in, HTML out.{% endHighlight %}
```

Source: Python

```
component('aardvark', 'highlight', highlight='Markdown',
          children='Write Markdown, get a site. Markdown in, HTML out.', attr={'onclick':
'''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

List

`{% list %}` is a **built-in** tag for ordered and unordered lists. A plain Markdown list still works for ordinary lists; reach for this tag when you want an icon bullet, an explicit size, or custom spacing. Give the items in `items`, pipe-separated (or as a JSON array); each item renders inline Markdown.

Use it as `{% list %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'list', …)`.

Demonstrations

Unordered (the default)

Pass `items` pipe-separated; each becomes a bulleted item.

Source: Markdown

```
{% list items='First item | Second item | Third item' %}
```

Source: Python

```
component('aardvark', 'list', items='First item | Second item | Third item')
```

Ordered

`type='ordered'` numbers the items instead of bulleting them.

Source: Markdown

```
{% list type='ordered' items='Clone the repo | Install deps | Run the build' %}
```

Source: Python

```
component('aardvark', 'list', type='ordered',
          items='Clone the repo | Install deps | Run the build')
```

Icon bullets

Set `icon` to a Tabler name to swap the bullet for an icon on every item.

Source: Markdown

```
{% list icon='circle-check' items='Tests pass | Build is green | Docs updated' %}
```

Source: Python

```
component('aardvark', 'list', icon='circle-check',
          items='Tests pass | Build is green | Docs updated')
```

Size, spacing, padding, center

`size` (`xs`–`xl`) sets the text size, `spacing` the gap between items, `withPadding` indents the list from its container, and `center` vertically centers each item against its bullet. Items render inline Markdown, so `**bold**`, `` `code` ``, and `[links]` all format.

Source: Markdown

```
{% list size='lg' spacing='md' withPadding=true center=true items='A **bold** item | An item
with `code` | A [linked](https://mantine.dev) item' %}
```

Source: Python

```
component('aardvark', 'list', size='lg', spacing='md', withPadding=True, center=True,
          items='A **bold** item | An item with `code` | A [linked](https://mantine.dev)
item')
```

JSON array items

When an item contains a literal pipe, pass `items` as a JSON array instead of a pipe-separated string.

Source: Markdown

```
{% list items='["Pipe in a | value", "A plain item", "Another item"]' %}
```

Source: Python

```
component('aardvark', 'list', items='["Pipe in a | value", "A plain item", "Another item"]')
```

With other components

Items render inline Markdown, so `` `code` `` and `[links]` format inside each item.

To compose items out of other built-in tags (Code, Mark, ...), build the item strings in Python and pass them as a list — the call form lets you mix tag output into each item.

Source: Markdown

```
{% list icon='arrow-right' items='Run `vark build` | Ship a [static
site](https://aardvark.dev) | Watch it **just work**' %}
```

Source: Python

```
{%
run = "Run " + component('aardvark', 'code', children='vark build')
ship = "Ship a " + component('aardvark', 'mark', color='lime', children='static site')
page.print(component('aardvark', 'list', icon='arrow-right', items=[run, ship]))
%}
```

Attributes

Attribute	Valid values	Description
`items`	Pipe-separated string (`First | Second`) or a JSON array; Python also accepts a list	The list items. Each renders inline Markdown. Empty entries are dropped.
`type`	`unordered` (default) `ordered`	Bulleted or numbered list.
`size`	`xs` `sm` `md` `lg` `xl`	Text size.
`spacing`	A Mantine size token (`xs`–`xl`)	Gap between items.
`withPadding`	`true` / `false` (default `false`)	Indent the list from its left edge.
`center`	`true` / `false` (default `false`)	Vertically center each item against its bullet/icon.
`icon`	A Tabler icon name (e.g. `circle-check`)	Used as the bullet for every item.

Items render inline Markdown, so `**bold**`, `` `code` ``, and `[links]` all format.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="List"]`, or through the Mantine Styles API classes:

```
/* Every rendered List carries this island marker */
[data-aardvark-island="List"] { }

/* Mantine Styles API classes */
.mantine-List-root { }
.mantine-List-item { }
.mantine-List-itemWrapper { }
.mantine-List-itemLabel { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% list items='First item | Second item | Third item' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'list', items='First item | Second item | Third item',
attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Mark

`{% mark %}` is a **built-in** tag for highlighted text — a tinted `<mark>` run you drop mid-sentence to draw the eye to a phrase. It's always inline, so it flows with the surrounding text. The highlighted text is the block body (inline Markdown) or a plain-text `text` param.

Use it as `{% mark %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'mark', …)`.

Demonstrations

Inline highlight

Drop `{% mark %}` mid-sentence to tint a phrase; it flows with the surrounding text.

Aardvark turns

Markdown

into a static site.

Source: Markdown

```
Aardvark turns {% mark %}Markdown{% endMark %} into a static site.
```

Source: Python

```
component('aardvark', 'mark', children='Markdown')
```

Color

`color` takes any theme color; it defaults to yellow.

yellow

lime

cyan

pink

Source: Markdown

```
{% mark %}yellow{% endMark %} {% mark color='lime' %}lime{% endMark %} {% mark color='cyan'
%}cyan{% endMark %} {% mark color='pink' %}pink{% endMark %}
```

Source: Python

```
component('aardvark', 'mark', children='yellow')
component('aardvark', 'mark', color='lime', children='lime')
component('aardvark', 'mark', color='cyan', children='cyan')
component('aardvark', 'mark', color='pink', children='pink')
```

Markdown content and the text shortcut

The block body renders inline Markdown, so you can format inside the run. A `text` param is the plain-text shortcut.

A **bold** word and a `token`

Source: Markdown

```
{% mark color='yellow' %}A **bold** word and a `token`{% endMark %}
```

Source: Python

```
component('aardvark', 'mark', color='yellow', children='A **bold** word and a `token`')
# plain-text shortcut:
component('aardvark', 'mark', color='yellow', text='A plain phrase')
```

With other components

Because Mark is inline, it nests cleanly inside other built-ins — here it tints a word inside a List item and a Blockquote.

The build is

green

when every link resolves.

Source: Markdown

```
{% blockquote icon='highlight' %}
The build is {% mark color='lime' %}green{% endMark %} when every link resolves.
{% endBlockquote %}
```

Source: Python

```
body = ("The build is "
        + component('aardvark', 'mark', color='lime', children='green')
        + " when every link resolves.")
component('aardvark', 'blockquote', icon='highlight', children=body)
```

Mark vs Highlight

Use `{% mark %}` for a **fixed phrase** you wrap by hand. To highlight **every occurrence** of a substring inside a longer passage, reach for Highlight instead — it scans the text and marks each match for you.

Attributes

Attribute	Valid values	Description
`text`	Any string	The text, used when there's no block body (plain text).
`color`	Any theme color (e.g. `lime`, `cyan`, `pink`)	Highlight tint. Defaults to yellow.

The highlighted text is the block body (inline Markdown — `**bold**`, `` `code` ``, `[links]`) or the `text` param.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Mark"]`, or through the Mantine Styles API classes:

```
/* Every rendered Mark carries this island marker */
[data-aardvark-island="Mark"] { }

/* Mantine Styles API classes */
.mantine-Mark-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Aardvark turns

Markdown

into a static site.

Source: Markdown

```
Aardvark turns {% mark attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Markdown{% endMark %} into a static site.
```

Source: Python

```
component('aardvark', 'mark', children='Markdown', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Table

`{% table %}` is a **built-in** tag for data tables — the Mantine Table element. A plain Markdown table still works; reach for this tag when you want striping, borders, row hover, a caption, or a sticky header. Give the header cells in `head` (pipe-separated) and the body in `rows` (rows separated by `;` or newlines, cells by `|`, or a JSON array of arrays); each cell renders inline Markdown.

Use it as `{% table %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'table', …)`.

Demonstrations

A basic table needs only `head` and `rows`:

Source: Markdown

```
{% table head='Name | Role | Status' rows='Ada | Engineer | Active ; Grace | Admiral | Active' %}
```

Source: Python

```
component('aardvark', 'table',
          head='Name | Role | Status',
          rows='Ada | Engineer | Active ; Grace | Admiral | Active')
```

Striping and hover

`striped` shades alternate rows (pass `odd` or `even` to choose which) and `highlightOnHover` lights up the row under the cursor:

Source: Markdown

```
{% table striped=true highlightOnHover=true head='Package | Version' rows='aardvark | 0.1.10 ; mantine | 9.x ; react | 19' %}
```

Source: Python

```
component('aardvark', 'table',
          striped=True, highlightOnHover=True,
          head='Package | Version',
          rows='aardvark | 0.1.10 ; mantine | 9.x ; react | 19')
```

Borders

Toggle the outer border (`withTableBorder`), column dividers (`withColumnBorders`), and row dividers (`withRowBorders`, on by default — set `false` to remove them):

Source: Markdown

```
{% table withTableBorder=true withColumnBorders=true head='Method | Path' rows='GET | /pets
; POST | /pets ; DELETE | /pets/{id}' %}
```

Source: Python

```
component('aardvark', 'table',
          withTableBorder=True, withColumnBorders=True,
          head='Method | Path',
          rows='GET | /pets ; POST | /pets ; DELETE | /pets/{id}')
```

Caption and inline Markdown

Add a `caption` (set `captionSide` to `top` or `bottom`, the default). Cells render inline Markdown, so `` `code` `` and `**bold**` format:

Source: Markdown

```
{% table caption='Build stats' captionSide='bottom' withTableBorder=true head='Stage | Time'
rows='`render` | **2.4s** ; `bundle` | 0.3s' %}
```

Source: Python

```
component('aardvark', 'table',
          caption='Build stats', captionSide='bottom', withTableBorder=True,
          head='Stage | Time',
          rows='`render` | **2.4s** ; `bundle` | 0.3s')
```

Spacing, width, and JSON rows

`horizontalSpacing` / `verticalSpacing` size the cell padding with Mantine tokens, and `minWidth` sets the width below which the table scrolls horizontally. Pass `rows` as a JSON array of arrays when a cell contains a literal `|` or `;`:

Source: Markdown

```
{% table horizontalSpacing='xl' verticalSpacing='md' minWidth='30rem' head='Key | Default'
rows='[["timeout", "`30 ; per call`"], ["retries", "3"]]' %}
```

Source: Python

```
component('aardvark', 'table',
          horizontalSpacing='xl', verticalSpacing='md', minWidth='30rem',
          head='Key | Default',
          rows='[["timeout", "`30 ; per call`"], ["retries", "3"]]')
```

Sticky header

`stickyHeader` pins the header while the body scrolls; `maxHeight` caps the scroll window (px, default 320) and `stickyHeaderOffset` pushes the pinned header down past a fixed navbar:

Source: Markdown

```
{% table stickyHeader=true maxHeight=140 striped=true head='Build | Result' rows='1.0.0 |
pass ; 1.0.1 | pass ; 1.0.2 | fail ; 1.0.3 | pass ; 1.0.4 | pass ; 1.0.5 | pass' %}
```

Source: Python

```
component('aardvark', 'table',
          stickyHeader=True, maxHeight=140, striped=True,
          head='Build | Result',
          rows='1.0.0 | pass ; 1.0.1 | pass ; 1.0.2 | fail ; 1.0.3 | pass ; 1.0.4 | pass ;
1.0.5 | pass')
```

With other components

Build a table from a loop in Python, or pass Text and Title inside a containing layout. Here the rows come from a Python list, joined into the `;/|` shorthand:

Release matrix

Source: Markdown

```
{% title order=4 mb='sm' %}Release matrix{% endTitle %}
{% table withTableBorder=true striped=true head='Version | Channel' rows='0.1.8 | stable ;
0.1.9 | stable ; 0.1.10 | latest' %}
```

Source: Python

```
releases = [('0.1.8', 'stable'), ('0.1.9', 'stable'), ('0.1.10', 'latest')]
rows = ' ; '.join(f'{v} | {c}' for v, c in releases)

component('aardvark', 'title', children='Release matrix', order=4, mb='sm')
component('aardvark', 'table',
          withTableBorder=True, striped=True,
          head='Version | Channel', rows=rows)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description

`head`	String, pipe-separated (`A | B | C`)	Header cells; each renders inline Markdown. Empty omits the `<thead>`.
`rows`	String, or JSON array of arrays	Body rows — rows split on `;` or newlines, cells on `|`; or `[["a","b"],…]`. Each cell renders inline Markdown.
`caption`	String	A caption rendered above or below the table.
`captionSide`	`top`, `bottom`	Which side the caption sits on. Default `bottom`.
`striped`	`true`, `false`, `odd`, `even`	Shade alternate rows. A bare `striped` flag or `true` stripes odd rows; `odd` / `even` choose which. Default `false`.
`highlightOnHover`	`true`, `false`	Highlight the row under the cursor. Default `false`.
`withTableBorder`	`true`, `false`	Outer border around the whole table. Default `false`.
`withColumnBorders`	`true`, `false`	Vertical dividers between columns. Default `false`.
`withRowBorders`	`true`, `false`	Horizontal dividers between rows. Default `true`; set `false` to remove them.
`stickyHeader`	`true`, `false`	Pin the header while the table body scrolls. Default `false`.
`stickyHeaderOffset`	Integer (px)	Offset for the sticky header, e.g. to clear a fixed navbar.
`maxHeight`	Integer (px)	Cap the sticky-header scroll window height. Default 320.
`horizontalSpacing`	Mantine size token (`xs`–`xl`) or CSS value	Horizontal cell padding.
`verticalSpacing`	Mantine size token (`xs`–`xl`) or CSS value	Vertical cell padding.

`minWidth`	CSS width (`30rem`, `600px`)	Minimum width before the table scrolls horizontally.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Table"]`, or through the Mantine Styles API classes. (Table uses element-named parts rather than a `-root`.) The relevant classes:

```
/* Every rendered Table carries this island marker */
[data-aardvark-island="Table"] { }

/* Mantine Styles API classes */
.mantine-Table-table { }
.mantine-Table-thead { }
.mantine-Table-tbody { }
.mantine-Table-tr { }
.mantine-Table-th { }
.mantine-Table-td { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Source: Markdown

```
{% table head='Name | Role | Status' rows='Ada | Engineer | Active ; Grace | Admiral |
Active' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
```

Source: Python

```
component('aardvark', 'table',
          head='Name | Role | Status',
          rows='Ada | Engineer | Active ; Grace | Admiral | Active',
          attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Text

`{% text %}` is a **built-in** tag for styled text — every Mantine Text capability (weight, size, color, alignment, transform, gradient, truncation, …) exposed as a single tag with no setup. Reach for it when plain Markdown emphasis isn't enough: a colored lead paragraph, an uppercase label, a gradient heading, a clamped excerpt.

Use it as `{% text %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'text', …)`.

Demonstrations

Content

Give the text inline with `text`, or as the block body. The block body is the richer form — it renders **inline Markdown** (`**bold**`, `*italic*`, `` `code` ``, and `[links](/)`), so you can format inside the styled run. Bare URLs in the block body auto-link, and the typographer applies (straight quotes → curly, `--` → en-dash). The inline `text` shortcut, by contrast, is plain text: no Markdown, no auto-linking, and no typography substitutions.

Inline content (plain).

Block content with **bold** and a `code` span.

Source: Markdown

```
{% text text='Inline content (plain).' %}
{% text c='grape' %}Block content with **bold** and a `code` span.{% endText %}
```

Source: Python

```
component('aardvark', 'text', text='Inline content (plain).')
component('aardvark', 'text', children='Block content with **bold** and a `code` span.',
c='grape')
```

Typography

Set the weight (`fw`), size (`size` or `fz`), style (`fs`), decoration (`td`), transform (`tt`), alignment (`ta`), font family (`ff`), line height (`lh`), and letter spacing (`lts`):

Bold

Italic

Underlined

Spaced caps

Extra large monospace

Centered

Source: Markdown

```
{% text fw=700 %}Bold{% endText %}
{% text fs='italic' %}Italic{% endText %}
{% text td='underline' %}Underlined{% endText %}
{% text tt='uppercase' lts='0.1em' %}Spaced caps{% endText %}
{% text size='xl' ff='monospace' %}Extra large monospace{% endText %}
{% text ta='center' %}Centered{% endText %}
```

Source: Python

```
component('aardvark', 'text', children='Bold', fw=700)
component('aardvark', 'text', children='Italic', fs='italic')
component('aardvark', 'text', children='Underlined', td='underline')
component('aardvark', 'text', children='Spaced caps', tt='uppercase', lts='0.1em')
component('aardvark', 'text', children='Extra large monospace', size='xl', ff='monospace')
component('aardvark', 'text', children='Centered', ta='center')
```

Color

`c` takes any theme color, the special `dimmed`, or any CSS color value:

Primary

Red

Dimmed secondary text

Source: Markdown

```
{% text c='primary' fw=600 %}Primary{% endText %}
{% text c='red' %}Red{% endText %}
{% text c='dimmed' %}Dimmed secondary text{% endText %}
```

Source: Python

```
component('aardvark', 'text', children='Primary', c='primary', fw=600)
component('aardvark', 'text', children='Red', c='red')
component('aardvark', 'text', children='Dimmed secondary text', c='dimmed')
```

Gradient

Set `variant='gradient'` and any of `gradientFrom`, `gradientTo`, `gradientDeg` for gradient text — pair it with a large `fz` and a bold `fw` for a hero heading:

Gradient heading

Source: Markdown

```
{% text variant='gradient' gradientFrom='indigo' gradientTo='cyan' gradientDeg=45
fz='2.5rem' fw=900 %}Gradient heading{% endText %}
```

Source: Python

```
component('aardvark', 'text', children='Gradient heading',
          variant='gradient', gradientFrom='indigo', gradientTo='cyan',
          gradientDeg=45, fz='2.5rem', fw=900)
```

Truncation

`truncate` clips to a single line with an ellipsis (`truncate='start'` clips the start instead); `lineClamp` clips to a fixed number of lines. Both need a width to act on — `maw` caps it here:

This sentence is far too long to fit on one short line and will be cut off.

This longer passage is clamped to two lines: anything past the second line is hidden behind an ellipsis so cards and previews stay tidy and the same height.

Source: Markdown

```
{% text truncate maw='20rem' %}This sentence is far too long to fit on one short line and
will be cut off.{% endText %}
{% text lineClamp=2 maw='20rem' %}This longer passage is clamped to two lines: anything past
the second line is hidden behind an ellipsis.{% endText %}
```

Source: Python

```
component('aardvark', 'text',
          children='This sentence is far too long to fit on one short line and will be cut
off.',
          truncate=True, maw='20rem')
component('aardvark', 'text',
          children='This longer passage is clamped to two lines: anything past the second
line is hidden behind an ellipsis.',
          lineClamp=2, maw='20rem')
```

Inline placement

By default `{% text %}` is a **block** — its own paragraph. To style a run of text **inside a sentence**, add `span` (it renders as an inline `<span>`, so it doesn't break the paragraph):

Ship it when the badge turns **green** — never while it's **red**.

Source: Markdown

```
Ship it when the badge turns {% text span=true c='green' fw=700 %}green{% endText %} — never
while it's {% text span=true c='red' fw=700 %}red{% endText %}.
```

Source: Python

```
component('aardvark', 'text', children='green', span=True, c='green', fw=700)
component('aardvark', 'text', children='red', span=True, c='red', fw=700)
```

With other components

Set a Title above a `{% text %}` standfirst, or use a `span` run to highlight a word inside ordinary prose. Build a legend from a Python loop:

Status key

Passing · **Failing** · Skipped

Source: Markdown

```
{% title order=4 mb='xs' %}Status key{% endTitle %}
{% text span=true c='green' fw=700 %}Passing{% endText %}
{% text span=true c='red' fw=700 %} · Failing{% endText %}
{% text span=true c='dimmed' %} · Skipped{% endText %}
```

Source: Python

```
component('aardvark', 'title', children='Status key', order=4, mb='xs')
for label, color in [('Passing', 'green'), (' · Failing', 'red'), (' · Skipped', 'dimmed')]:
    component('aardvark', 'text', children=label, span=True, c=color,
              fw=700 if color != 'dimmed' else None)
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`text`	String	The text, when not using the block body. Plain (no Markdown).
`size`	`xs`–`xl`, or any CSS size	Font size.

`fz`	`xs`–`xl`, or any CSS size (`1.25rem`)	Font size (alias of `size`).
`fw`	`bold`, `normal`, or a number (`700`)	Font weight.
`fs`	`italic`, `normal`	Font style.
`td`	`underline`, `line-through`, `none`	Text decoration.
`tt`	`uppercase`, `capitalize`, `lowercase`, `none`	Text transform.
`ta`	`left`, `center`, `right`, `justify`	Text alignment.
`c`	Theme color, `dimmed`, or any CSS color	Text color.
`ff`	`monospace`, `heading`, `text`, or a CSS family	Font family.
`lh`	Number or any CSS line-height	Line height.
`lts`	Any CSS letter-spacing (`0.1em`)	Letter spacing.
`variant`	`text` (default), `gradient`	Render style; `gradient` enables the gradient props.
`gradientFrom`	Theme color or CSS color	Gradient start color (with `variant='gradient'`).
`gradientTo`	Theme color or CSS color	Gradient end color (with `variant='gradient'`).
`gradientDeg`	Number (degrees)	Gradient angle (with `variant='gradient'`).

`truncate`	`true`, `start`, `end`	Single-line ellipsis — bare flag / `true` clips the end, `start` clips the start.
`lineClamp`	Positive integer	Clip to this many lines.
`inline`	`true`, `false`	Set line height to 1 (tight — to vertically center a single line). A typography tweak only; does **not** make the text flow inline — use `span` for that. Default `false`.
`inherit`	`true`, `false`	Inherit font properties from the parent (e.g. inside a heading). Default `false`.
`span`	`true`, `false`	Render inline (a `<span>`) so it sits mid-sentence. Otherwise the tag is a block paragraph. Default `false`.
`id`	String	HTML `id` on the rendered element, for CSS / JS selectors.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	Mantine size token or any CSS value	Margin (all, top, bottom, left, right, horizontal, vertical).
`p`, `pt`, `pb`, `pl`, `pr`, `px`, `py`	Mantine size token or any CSS value	Padding (all, top, bottom, left, right, horizontal, vertical).
`bg`	Theme color or any CSS color	Background color.
`opacity`	`0`–`1`	Opacity.
`w`, `h`, `miw`, `mih`, `maw`, `mah`	Mantine size token or any CSS value	Width / height and their min / max.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Text"]`, or through the Mantine Styles API classes:

```
/* Every rendered Text carries this island marker */
[data-aardvark-island="Text"] { }
```

```
/* Mantine Styles API classes */
.mantine-Text-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Block content with **bold** and a `code` span.

Source: Markdown

```
{% text c='grape' attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Block content with **bold** and a `code` span.{% endText %}
```

Source: Python

```
component('aardvark', 'text', children='Block content with **bold** and a `code` span.',
c='grape', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Title

`{% title %}` is a **built-in** tag for headings — the Mantine Title element (`<h1>`–`<h6>`) with explicit control over the level and its styling. A plain Markdown `#` heading still works for ordinary headings; reach for this tag when you want a heading with a specific color, alignment, transform, or a visual size that differs from its level.

Use it as `{% title %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'title', …)`.

Demonstrations

Order

`order` sets the heading level, `1`–`6` (`<h1>` … `<h6>`). It controls both the semantics and the default visual size. Out-of-range values fall back to Mantine's default:

Order 1

Order 3

Order 6

Source: Markdown

```
{% title order=1 %}Order 1{% endTitle %}
{% title order=3 %}Order 3{% endTitle %}
{% title order=6 %}Order 6{% endTitle %}
```

Source: Python

```
for n in (1, 3, 6):
    component('aardvark', 'title', children=f'Order {n}', order=n)
```

Visual size

Set `size` to detach the visual size from the level — a small `<h2>` for SEO without a giant heading, say. It takes a heading token (`h1`–`h6`), a Mantine size (`xs`–`xl`), or any CSS size:

An h2 that looks like an h5

An h2 at 3rem

Source: Markdown

```
{% title order=2 size='h5' %}An h2 that looks like an h5{% endTitle %}
{% title order=2 size='3rem' %}An h2 at 3rem{% endTitle %}
```

Source: Python

```
component('aardvark', 'title', children='An h2 that looks like an h5', order=2, size='h5')
component('aardvark', 'title', children='An h2 at 3rem', order=2, size='3rem')
```

Color, align, transform, weight

The usual typography props apply: `c` (color), `ta` (align), `tt` (transform), and `fw` (weight). `lh` and `lts` tune line height and letter spacing:

Centered grape heading

Bold spaced caps

Source: Markdown

```
{% title order=3 c='grape' ta='center' %}Centered grape heading{% endTitle %}
{% title order=3 tt='uppercase' fw=900 lts='0.1em' %}Bold spaced caps{% endTitle %}
```

Source: Python

```
component('aardvark', 'title', children='Centered grape heading',
          order=3, c='grape', ta='center')
component('aardvark', 'title', children='Bold spaced caps',
          order=3, tt='uppercase', fw=900, lts='0.1em')
```

Content and spacing

Give the heading inline with `text`, or as the block body. The block body renders inline Markdown (`**bold**`, `` `code` ``, `[links]`). Margin props (`mt`, `mb`, ...) space the heading from its neighbours:

Inline heading

Heading with **emphasis** and a `token`

Source: Markdown

```
{% title order=4 text='Inline heading' %}
{% title order=4 mt='lg' mb='xs' %}Heading with **emphasis** and a `token`{% endTitle %}
```

Source: Python

```
component('aardvark', 'title', order=4, text='Inline heading')
component('aardvark', 'title', children='Heading with **emphasis** and a `token`',
          order=4, mt='lg', mb='xs')
```

With other components

Pair a title with Text for a lead paragraph, or sit one above a Table as a section header. Here a heading and a dimmed standfirst form a section intro:

Quick start

Install the binary, scaffold a site, and run `vark build`.

Source: Markdown

```
{% title order=3 mb='xs' %}Quick start{% endTitle %}
{% text c='dimmed' %}Install the binary, scaffold a site, and run `vark build`.{% endText %}
```

Source: Python

```
component('aardvark', 'title', children='Quick start', order=3, mb='xs')
component('aardvark', 'text',
          children='Install the binary, scaffold a site, and run `vark build`.',
          c='dimmed')
```

Attributes

Omit any attribute to take its Mantine default.

Attribute	Valid values	Description
`text`	String	The heading text, when not using the block body. Plain text (no Markdown).
`order`	`1`–`6`	Heading level (`<h1>`–`<h6>`); sets semantics and default size. Out-of-range falls back to the default.
`size`	`h1`–`h6`, `xs`–`xl`, or any CSS size	Visual size, independent of `order`.
`c`	Theme color, `dimmed`, or any CSS color	Text color.
`ta`	`left`, `center`, `right`, `justify`	Text alignment.
`tt`	`uppercase`, `capitalize`, `lowercase`, `none`	Text transform.

`fw`	`bold`, `normal`, or a number (`700`)	Font weight.
`lh`	Number or any CSS line-height	Line height.
`lts`	Any CSS letter-spacing (`0.1em`)	Letter spacing.
`id`	String	HTML `id` on the rendered heading, for CSS / JS selectors and anchor links.
`m`, `mt`, `mb`, `ml`, `mr`, `mx`, `my`	Mantine size token or any CSS value	Margin (all, top, bottom, left, right, horizontal, vertical).

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Title"]`, or through the Mantine Styles API classes:

```
/* Every rendered Title carries this island marker */
[data-aardvark-island="Title"] { }

/* Mantine Styles API classes */
.mantine-Title-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

Order 1

Source: Markdown

```
{% title order=1 attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}Order 1{% endTitle %}
```

Source: Python

```
component('aardvark', 'title', children='Order 1', order=1, attr={'onclick': '''
```

```
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

Typography

`{% typography %}` is a **built-in** wrapper that applies Mantine's article prose styles to whatever it contains — headings, paragraphs, lists, tables, blockquotes, code, and links all get consistent spacing and type. It's the Mantine Typography component. Reach for it around a chunk of generated or pasted HTML, or a block of Markdown, that would otherwise be unstyled. The block body is the prose; close with `{% endTypography %}`.

Use it as `{% typography %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'typography', …)`.

You usually don't need this

Page content is **already** styled — Aardvark applies its own article prose styles to every page (the `.aardvark-content` surface), so your headings, lists, tables, code, and links look right with no wrapper at all. `{% typography %}` is included for **completeness**; reach for it only to style a block of prose or raw HTML that sits **outside** that flow — content you render **inside a portaled component** (a modal, dialog, or popover body) that React mounts outside the page's content container. In ordinary page Markdown it's a no-op.

Demonstrations

The provider takes no styling props of its own — it styles its children. Wrap a block of Markdown and the headings, paragraphs, lists, blockquotes, code, and links all pick up the article styles:

A styled article

This paragraph, the heading above, and everything below pick up Mantine's prose styles — including **emphasis**, `inline code`, and links.

- A list item
- Another item

> A blockquote, styled to match.

Source: Markdown

```
{% typography %}
## A styled article

This paragraph, the heading above, and everything below pick up Mantine's prose
styles — including **emphasis**, `inline code`, and [links](https://mantine.dev).

- A list item
- Another item

> A blockquote, styled to match.
```

```
{% endTypography %}
```

Source: Python

```
prose = '''
## A styled article

This paragraph, the heading above, and everything below pick up Mantine's prose
styles — including **emphasis**, `inline code`, and [links](https://mantine.dev).

- A list item
- Another item

> A blockquote, styled to match.
'''
component('aardvark', 'typography', children=prose)
```

With other components

The provider styles plain prose; for finer control over a single run of text, reach for Text or Title instead. A common pattern is a standalone Title above a `{% typography %}` block of body copy:

Release notes

What's new in this build:

1. Faster incremental rebuilds.
2. A new `{% table %}` tag.

See the changelog for the full list.

Source: Markdown

```
{% title order=3 mb='sm' %}Release notes{% endTitle %}
{% typography %}
What's new in this build:

1. Faster incremental rebuilds.
2. A new {% table %} tag.

See the [changelog](https://mantine.dev) for the full list.
{% endTypography %}
```

Source: Python

```
component('aardvark', 'title', children='Release notes', order=3, mb='sm')
component('aardvark', 'typography', children='''
What's new in this build:

1. Faster incremental rebuilds.
```

```
2. A new table tag.

See the [changelog](https://mantine.dev) for the full list.
''')
```

Attributes

`{% typography %}` takes no attributes — the block body is the prose it styles. It is rendered as Markdown, so anything you can write on a normal page works inside it.

Attribute	Valid values	Description
(body)	Markdown or raw HTML	The prose to style; everything between the tag and `{% endTypography %}`.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="Typography"]`, or through the Mantine Styles API classes:

```
/* Every rendered Typography carries this island marker */
[data-aardvark-island="Typography"] { }

/* Mantine Styles API classes */
.mantine-Typography-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes — including event handlers — straight onto the rendered element, so you can wire DOM behavior the tag does not expose. The handler can be a full multi-line script, not just one expression — this one logs the value to the console and shows it in an alert:

A heading

A paragraph with a link and `inline code`.

Source: Markdown

```
{% typography attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''} %}
## A heading
```

```
A paragraph with a [link](/) and `inline code`.
{% endTypography %}
```

Source: Python

```
component('aardvark', 'typography', children='''
## A heading

A paragraph with a [link](/) and `inline code`.
''', attr={'onclick': '''
const value = this.innerText;
console.log('attr demo value:', value);
alert(value);
'''})
```

VisuallyHidden

`visuallyhidden` is the standard **screen-reader-only** accessibility primitive: it hides its content visually — it's gone from the layout — while keeping it in the accessibility tree, so screen readers still announce it. By design it renders **no visible output**: the content is positioned off-screen but stays available to assistive technology. Use it for labels, captions, and context that assistive tech needs but that would be redundant or cluttering on screen — for example extra wording for an icon-only button, or a heading that gives screen-reader users the structure a sighted user gets from layout. It takes no styling options. Close the block with `{% endVisuallyhidden %}` (one capital V).

Use it as `{% visuallyhidden %}…{% endVisuallyhidden %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'visuallyhidden', …)`.

Demonstrations

The block body is the screen-reader-only content. It produces nothing visible on the page — open it in a screen reader, or inspect the DOM, to find the hidden run of text.

Preview (there is no visual difference; the hidden text is announced by assistive tech only):

Visible text: "Here is some text"

and this part is only for screen readers

.

Source: Markdown

```
Visible text: "Here is some text"
{% visuallyhidden %}and this part is only for screen readers{% endVisuallyhidden %}.
```

Source: Python

```
'Visible text: "Here is some text" ' + component(
    'aardvark', 'visuallyhidden',
    children='and this part is only for screen readers')
```

With other components

The classic use is an **icon-only button**: the button shows only the icon, while a `visuallyhidden` run gives screen-reader users the accessible name. The button below shows just the trash icon, but a screen reader announces "Delete this item".

Preview:

Delete this item

Source: Markdown

```
<button>
  {% icon 'fa-solid fa-trash' %}
  {% visuallyhidden %}Delete this item{% endVisuallyhidden %}
</button>
```

Source: Python

```
label = component('aardvark', 'visuallyhidden', children='Delete this item')
icon = component('aardvark', 'icon', 'fa-solid fa-trash')
'<button>' + icon + label + '</button>'
```

Attributes

`visuallyhidden` takes no styling options — it's a single-purpose accessibility wrapper.

Attribute	Valid values	Description
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered span if you need a hook.

CSS Selectors

Target the rendered element through its island marker, `[data-aardvark-island="VisuallyHidden"]`, or through the Mantine Styles API classes:

```
/* Every rendered VisuallyHidden carries this island marker */
[data-aardvark-island="VisuallyHidden"] { }

/* Mantine Styles API classes */
.mantine-VisuallyHidden-root { }
```

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered element. A `visuallyhidden` element is off-screen, so a click handler isn't practically reachable — instead, `attr` is useful for `data-*` hooks, ARIA, or an `id` that scripts and tests can target on the (otherwise invisible) node:

Visible text

and this part is only for screen readers

. (Inspect the hidden span in DevTools to see the forwarded `data-testid`.)

Source: Markdown

```
Visible text {% visuallyhidden attr={'data-testid': 'sr-only-note'} %}and this part is only
for screen readers{% endVisuallyhidden %}.
```

Source: Python

```
'Visible text ' + component(
    'aardvark', 'visuallyhidden',
    children='and this part is only for screen readers',
    attr={'data-testid': 'sr-only-note'}) + '.'
```

Community Components

Most built-in tags wrap a **Mantine core** component, and the Aardvark Extras are written in-house. **Community Components** are different: each one wraps a third-party **Mantine extension** from the Mantine extensions catalog — a widget written by a community member, not by the Mantine core team or by Aardvark. Extensions listed under **Bundled components** expose a built-in `{% tag %}` you can drop straight into Markdown.

Provenance & inclusion gates

These projects are **not ours**. Each is independently authored and maintained by its original author, and we bundle one only on these terms:

- **Permissively licensed.** Every bundled extension is MIT-licensed (or similarly permissive). Commercial use alone is not enough; licenses outside the project's permissive allowlist do not ship as community tags.
- **Compatible as published.** The package's published peer and direct dependencies must support the Mantine and React majors used by Aardvark. Overrides and substitute implementations do not count as compatibility.
- **Credited to its author.** Each per-tag page names the original author and links to the source project, so credit (and the license terms) travel with the component.
- **Bundled from npm.** The underlying package is pulled straight from npm at the version we wrap; we don't fork or vendor a modified copy.

The catalog is kept current by an **automated monitor** — the `refresh-community-extensions` workflow (and its companion skill) watches the upstream Mantine extensions list and opens a PR as new extensions pass the license, framework, and conversion-architecture gates or existing ones move. Rejected projects remain recorded in the source manifest so the monitor does not repeatedly propose them.

Bundled components

Each tag below wraps one community extension. Follow a tag's page for the author credit, license, the npm package it bundles, and live examples.

- Clock — an analog or digital clock face.
- LED — a glowing status indicator light.
- Spinner — extra loading-spinner styles beyond the core loader.
- Flip — a card that flips between a front and back face.
- Reflection — a mirrored reflection beneath its content.
- Parallax — content that shifts on scroll or pointer movement.
- Marquee — a continuously scrolling ticker row.

- TextAnimate — animated text reveals and transitions.
- BorderAnimate — an animated gradient border.
- Compare — a before/after image slider.
- Mask — masked/clipped content shapes.
- Scene — a 3D scene container.
- RingsProgress — nested concentric progress rings.
- SelectStepper — a stepper-style select control.
- QRCode — a rendered QR code for any value.
- Book — a page-turning book viewer.
- DepthSelect — a depth/layered select control.
- LensSelect — a magnifying-lens selector.
- JSONTree — a collapsible JSON tree viewer.
- ListViewTable — a list-style data table.
- Picker — a wheel/scroll value picker.
- Window — a draggable, resizable window frame.
- SplitPane — resizable split panels.
- Onboarding — a guided product tour overlay.
- Audio — a styled audio player.
- Video — a styled video player.
- ContextMenu — a right-click context menu.
- Lightbox — a full-screen image lightbox.
- DataTable — a feature-rich sortable, paginated data table.

Not bundled as Community Components

- **Mantine Map** — used only as an audited implementation dependency of Aardvark's native `{% map %}` block. It is deliberately not converted into a separate Community Component, and no `{% mantineMap %}` tag exists.
- BlockNote — compatible with the current framework, but excluded by the permissive-license policy because it is MPL-2.0 licensed.
- **FormBuilder** — excluded because the exact 0.1.28 registry record and published package.json omit license metadata (npm's MIT field appears only in versions 0.1.6 through 0.1.11), the repository has no LICENSE, and a README-only MIT statement does not meet the license gate. The package also peers and directly depends on Mantine 8 rather than Mantine 9.
- **Mantine React Table** — excluded because its published stable/beta lines target older Mantine majors, not Mantine 9.

- **Mantine Choropleth** — excluded because its published peers require Mantine 8; an npm override is not compatibility evidence.

Clock

`clock` is an analog clock. With no props it shows the live local time, ticking second by second; give it a `value` and `running=false` to freeze it on a fixed time, or a `timezone` to show another part of the world.

A **Community Component** — wraps Clock by **gfazioli**, **MIT** licensed, installed from npm as `@gfazioli/mantine-clock`.

Demonstrations

A bare `{% clock %}` renders a live analog clock showing the current local time.

```
{% clock %}
```

A fixed time

Pass a `value` (`"HH:MM"` or `"HH:MM:SS"`) with `running=false` to show a static time instead of the live clock.

```
{% clock value='10:09:36' running=false size='lg' %}
```

Timezones

Set `timezone` to an IANA timezone name to show the live time elsewhere.

```
{% group %}
{% clock timezone='America/New_York' size='md' %}
{% clock timezone='Europe/London' size='md' %}
{% clock timezone='Asia/Tokyo' size='md' %}
{% endGroup %}
```

Shape and second-hand behavior

`shape` (`circle` or `rounded-rect`) restyles the face, and `secondHandBehavior` (`smooth`, `tick`, or `tick-half`) controls how the second hand moves.

```
{% group %}
{% clock shape='circle' secondHandBehavior='smooth' size='md' %}
{% clock shape='rounded-rect' secondHandBehavior='tick' size='md' %}
{% endGroup %}
```

Arcs

Toggle `withSecondsArc`, `withMinutesArc`, and `withHoursArc` to draw progress arcs around the face.

```
{% clock withSecondsArc withMinutesArc withHoursArc size='lg' %}
```

With other components

A clock anchors a “current time” panel inside a card.

```
{% card title="Server time" %}
{% center %}
{% clock timezone='UTC' size='lg' withSecondsArc %}
{% endCenter %}
{% endCard %}
```

Server time

Attributes

Omit any attribute to take its default (a live local-time clock). Bare flags (e.g. `withSecondsArc`) become `=True`.

Attribute	Valid values	Description
`value`	`"HH:MM"` / `"HH:MM:SS"`	A fixed time to display (use with `running=false`).
`running`	`true` / `false`	Whether the clock ticks. Defaults to ticking; set `running=false` to freeze on `value`.
`timezone`	An IANA timezone name (e.g. `Asia/Tokyo`)	Show the live time for another zone. Defaults to local.
`size`	`xs`–`xl` or a number of px	Clock size.
`shape`	`circle`, `rounded-rect`	Face shape.
`secondHandBehavior`	`smooth`, `tick`, `tick-half`	How the second hand moves.
`withSecondsArc`	`true` / `false` (default `false`)	Draw the seconds progress arc.
`withMinutesArc`	`true` / `false` (default `false`)	Draw the minutes progress arc.

`withHoursArc`	`true` / `false` (default `false`)	Draw the hours progress arc.
`animateOnMount`	`true` / `false` (default `false`)	Play an entrance animation when the clock mounts.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

The clock takes no body.

CSS Selector

The clock exposes Mantine Styles API selectors you can target with `classNames` in your own React, or style directly:

Selector	Element
`root`	The clock root.
`clockFace`	The face background.
`hourTick` / `minuteTick`	The hour and minute tick marks.
`number` / `primaryNumber` / `secondaryNumber`	The numerals.
`hourHand` / `minuteHand` / `secondHand`	The three hands.
`centerDot`	The center hub.

Forward a class with `attr={…}` (see below) to scope a stylesheet rule to one instance.

Injecting Attributes

The `attr={…}` channel forwards raw HTML attributes straight onto the rendered clock root — use it for `id`, `class`, `data-*`, ARIA attributes, or anything else not exposed as a typed parameter.

Source: Markdown

```
{% clock timezone='UTC' attr={'class': 'office-clock', 'aria-label': 'UTC time'} %}
```

Source: Python

```
component('aardvark', 'clock', timezone='UTC',
          attr={'class': 'office-clock', 'aria-label': 'UTC time'})
```

ContextMenu

A **right-click context menu**: right-click the target and a floating menu of actions appears at the cursor. `label` is the visible target text; `items` is a JSON array describing the menu entries — actions and dividers. The menu opens on a real right-click after the page hydrates, so in a static screenshot you'll see only the target box; right-click it in a live browser to open the menu.

A **Community Component** — wraps mantine-contextmenu by **Ionut-Cristian Florescu**, **MIT** licensed, npm `mantine-contextmenu`.

Use it as `{% contextmenu %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'contextmenu', …)`.

Each entry in `items` is a small JSON object:

- **An action** — a `key`, a `title`, plus optional `href` (navigates there when picked on a static page), `color` (e.g. `red` for a destructive action), and `icon` (a Tabler icon name, e.g. `copy`).
- **A divider** — `{"divider": true}`, a horizontal rule between groups.

Demonstrations

Actions, icons, and dividers

Each action takes a `key` and a `title`; add an `icon` (a Tabler icon name) for a leading glyph, a `color` to tint a destructive action, and a `{"divider": true}` entry to rule off a group.

Right-click this card

Source: Markdown

```
{% contextmenu label='Right-click this card' items='[
  {"key": "copy", "title": "Copy to clipboard", "icon": "copy"},
  {"key": "download", "title": "Download", "icon": "download"},
  {"divider": true},
  {"key": "delete", "title": "Delete", "icon": "trash", "color": "red"}
]' %}
```

Source: Python

```
component('aardvark', 'contextmenu', label='Right-click this card', items='''[
  {"key": "copy", "title": "Copy to clipboard", "icon": "copy"},
  {"key": "download", "title": "Download", "icon": "download"},
  {"divider": true},
  {"key": "delete", "title": "Delete", "icon": "trash", "color": "red"}
]''')
```

Link actions

Give an action an `href` and picking it navigates there — the only meaningful side effect a static page can take. Relative paths and `http(s)://` links are allowed; unsafe schemes are rejected at build time.

Right-click for links

Source: Markdown

```
{% contextmenu label='Right-click for links' items='[
  {"key": "docs", "title": "Open the docs", "icon": "book", "href": "/"},
  {"key": "repo", "title": "View on GitHub", "icon": "brand-github", "href":
"https://github.com/"}
]' %}
```

Source: Python

```
component('aardvark', 'contextmenu', label='Right-click for links', items='''[
  {"key": "docs", "title": "Open the docs", "icon": "book", "href": "/"},
  {"key": "repo", "title": "View on GitHub", "icon": "brand-github", "href":
"https://github.com/"}
]''')
```

With other components

Because `items` is plain data, a Python caller can build the menu from a loop — for example mapping a list of pages to link actions — and render it under a Text heading:

Right-click for documentation sections

Right-click here

Source: Markdown

```
{% text fw='600' %}Right-click for documentation sections{% endText %}

{% contextmenu label='Right-click here' items='[
  {"key": "start", "title": "Getting started", "href": "/"},
  {"key": "components", "title": "Components", "href": "/"},
  {"key": "config", "title": "Configuration", "href": "/"}
]' %}
```

Source: Python

```
import json

pages = [
    ('start', 'Getting started', '/'),
    ('components', 'Components', '/'),
```

```
    ('config', 'Configuration', '/'),
]
items = json.dumps([{'key': k, 'title': t, 'href': u} for k, t, u in pages])

component('aardvark', 'text', fw='600',
          children='Right-click for documentation sections')
component('aardvark', 'contextmenu', label='Right-click here', items=items)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`label`	string	The visible right-click target text. Defaults to `Right-click here`.
`items`	JSON array string	The menu entries. Each is an action object (`key`, `title`, plus optional `href`, `color`, and a Tabler `icon` name) or a divider object (`{"divider": true}`).
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered target element. See Injecting Attributes.

CSS Selector

The right-click target carries the class `aardvark-contextmenu-target`; target it to restyle the surface authors see on the page:

```
.aardvark-contextmenu-target {
  background: var(--mantine-color-gray-0);
}
```

The floating menu itself is styled by the upstream `mantine-contextmenu` stylesheet, which the component pulls in automatically.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes onto the rendered target element — useful for `id`, `data-*`, ARIA attributes, or anything not exposed as a named param:

Right-click me

Source: Markdown

```
{% contextmenu label='Right-click me' attr={'id': 'demo-context', 'data-role':
```

```
 'menu-target'} items='[{"key": "x", "title": "Action"}]' %}
```

Source: Python

```
component('aardvark', 'contextmenu', label='Right-click me',
          attr={'id': 'demo-context', 'data-role': 'menu-target'},
          items='[{"key": "x", "title": "Action"}]')
```

Picker

`{% picker %}` is an **animated, iOS-style scroll/wheel picker**: a vertical wheel of options where the centred item is the selected one. Drag it, spin it with the mouse wheel, or arrow-key through it — the items scale, fade, and snap into place. Give it the options as `data` (a comma-separated list) and, optionally, a starting `defaultValue`.

A **Community Component** — wraps Picker by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-picker`.

Use it as `{% picker %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'picker', …)`.

Demonstrations

Basic options

Pass the options as `data` — a comma-separated list. The first item is selected by default; the centred row is the current value.

```
{% picker data='Red, Orange, Yellow, Green, Blue, Indigo, Violet' %}
```

A starting value

Set `defaultValue` to pre-select an item other than the first.

```
{% picker data='Mon, Tue, Wed, Thu, Fri, Sat, Sun' defaultValue='Wed' %}
```

Taller rows and more visible items

`itemHeight` sets each row's height in pixels and `visibleItems` how many rows show at once (use an odd number so one sits in the centre).

```
{% picker data='10, 20, 30, 40, 50, 60, 70, 80, 90' defaultValue='40' itemHeight=52
visibleItems=5 %}
```

Plain wheel (no band)

Turn off the highlight band and dividers with `withHighlight=false` and `withDividers=false` for a cleaner wheel, and switch off the wrap-around with `loop=false`.

```
{% picker data='Bronze, Silver, Gold, Platinum' withHighlight=false withDividers=false
loop=false %}
```

With other components

The picker is a self-contained widget, so it slots straight into any layout. Here a `paper` surface frames it with a `text` label above.

```
{% paper withBorder=true p='md' radius='md' %}
{% text fw=600 %}Pick a size{% endText %}

{% picker data='XS, S, M, L, XL, XXL' defaultValue='M' %}
{% endPaper %}
```

Pick a size

Attributes

Omit any attribute to take its default. Bare flags (e.g. `disabled`) become `=true`.

Attribute	Valid values	Description
`data`	comma-separated list, e.g. `'Red, Green, Blue'`	The options shown on the wheel.
`defaultValue`	one of the `data` values	The initially selected item (defaults to the first).
`itemHeight`	integer px (`40` default)	Height of each row.
`visibleItems`	odd integer (`3` default)	How many rows are visible at once.
`loop`	`true` (default) / `false`	Wrap around past the last item back to the first.
`animate`	`true` (default) / `false`	Animate the scroll to the selected value on mount.
`withHighlight`	`true` (default) / `false`	Show the highlight band over the selected row.
`withDividers`	`true` (default) / `false`	Show the dividers above and below the selected row.
`size`	`xs`–`xl`	Component size.
`label`	text	Accessible label for the picker.
`disabled`	`true` / `false` (default)	Disable all interaction.

`readOnly`	`true` / `false` (default)	Keep the appearance but block interaction.
`attr={…}`	an object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The picker renders inside an island wrapper you can target in custom CSS:

```
[data-aardvark-island="Picker"] {
  /* your overrides */
}
```

The package also exposes its own CSS variables on the picker root — `--picker-height`, `--picker-item-height`, `--picker-animation-duration`, `--picker-mask-height`, and more — which you can set through the `attr` style passthrough below.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (data attributes, ARIA, inline `style`) straight onto the rendered element — handy for hooks your own CSS or scripts key off, or for setting the package's CSS variables.

Source: Markdown

```
{% picker data='1, 2, 3, 4, 5' attr={'data-testid': 'qty-picker', 'style':
'--picker-mask-height: 40%'} %}
```

Source: Python

```
component('aardvark', 'picker', data='1, 2, 3, 4, 5', attr={'data-testid': 'qty-picker',
'style': '--picker-mask-height: 40%'})
```

Rings progress

A built-in tag for a **multi-ring (concentric)** progress indicator — several donut rings nested inside one another, each filling to its own value. Pass `rings` as a JSON array of `{value, color}` objects (one ring per object) and optionally set the overall `size`, the `gap` between rings, a center `label`, and flair like `glow`, `animate`, and `showValues`.

> **Not the same as `{% ringprogress %}`.** Aardvark also ships a singular Ring progress tag that wraps Mantine **core**'s `RingProgress` (one ring, multiple arcs). This `{% ringsprogress %}` tag is the plural community widget — multiple concentric rings — so the two never collide.

A **Community Component** — wraps RingsProgress by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-rings-progress`.

Demonstrations

Basic rings

Give each ring a `value` (0–100) and a `color`. Rings are drawn from the outside in.

```
{% ringsprogress
rings='[{"value":75,"color":"cyan"},{"value":50,"color":"orange"},{"value":25,"color":"grape
"}]' size=180 %}
```

Center label and values

Set `label` for text in the middle of the rings, and `showValues=true` to print each ring's value at the end of its arc.

```
{% ringsprogress rings='[{"value":80,"color":"teal"},{"value":55,"color":"indigo"}]'
size=200 label='68%' showValues=true %}
```

68%

Glow and animation

`glow=true` adds a neon glow; `animate=true` plays an entrance animation as the rings fill.

```
{% ringsprogress
rings='[{"value":90,"color":"lime"},{"value":60,"color":"pink"},{"value":30,"color":"violet"
}]' size=200 glow=true animate=true %}
```

Gap, thickness, and direction

`gap` sets the space between rings (px), `thickness` the stroke width, and `direction` flips the fill between `clockwise` (default) and `counterclockwise`. Arc ends are rounded by default; set `roundCaps=false` for square ends.

```
{% ringsprogress rings='[{"value":70,"color":"blue"},{"value":45,"color":"red"}]' size=180
gap=10 thickness=14 direction='counterclockwise' roundCaps=false %}
```

With other components

Drop a rings indicator inside a Card to build a compact stat tile.

```
{% card title="Quarterly goals" %}
{% ringsprogress
rings='[{"value":82,"color":"teal"},{"value":64,"color":"cyan"},{"value":40,"color":"indigo"
}]' size=160 label='Q3' %}
{% endCard %}
```

Quarterly goals

Q3

Attributes

Omit any attribute to take its default. Bare flags (e.g. `glow`) become `=True`.

Attribute	Valid values	Description
`rings`	**Required.** A JSON array of `{value, color}` objects	One ring per object; `value` is 0–100, `color` a Mantine color name or CSS color.
`size`	Integer (px)	Overall diameter of the outermost ring.
`thickness`	Integer (px)	Stroke width of each ring (overridable per-ring in the JSON).
`gap`	Integer (px)	Space between adjacent rings.
`roundCaps`	`true` / `false` (default `true`)	Round the ends of each arc; set `roundCaps=false` for square ends.
`animate`	`true` / `false` (default `false`)	Play an entrance animation as the rings fill.
`glow`	`true` / `false` (default `false`)	Add a neon glow effect.
`showValues`	`true` / `false` (default `false`)	Print each ring's value at the end of its arc.
`startAngle`	Integer (degrees)	Starting position of the arcs.

`direction`	`clockwise` (default) / `counterclockwise`	Fill direction.
`label`	String	Text shown in the center.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The component renders inside an island wrapper carrying `data-aardvark-island="RingsProgress"`, with the vendor's own class names on the SVG inside. Target the wrapper to position or space the widget:

```
[data-aardvark-island="RingsProgress"] {
  margin-inline: auto;
}
```

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes (ids, `data-*`, ARIA) straight onto the rendered root — handy for testing hooks or scripting.

Source: Markdown

```
{% ringsprogress rings='[{"value":60,"color":"cyan"}]' size=160 attr={'data-testid':
'cpu-rings', 'aria-label': 'CPU usage'} %}
```

Source: Python

```
component('aardvark', 'ringsprogress', rings='[{"value":60,"color":"cyan"}]', size=160,
          attr={'data-testid': 'cpu-rings', 'aria-label': 'CPU usage'})
```

Led

`led` is a small LED status indicator — a glowing dot you can drive on or off, tint, shape, and animate. Reach for it to show health, connection, or power state in a status panel.

A **Community Component** — wraps Led by **gfazioli**, **MIT** licensed, installed from npm as `@gfazioli/mantine-led`.

Demonstrations

A bare `{% led %}` renders a lit LED in the default style.

```
{% led %}
```

On and off

`value` drives the LED's state. Set `value=false` for an off LED, and `offColor` for the color it shows when off.

```
{% group %}
{% led value=true color='green' label='On' %}
{% led value=false offColor='gray' label='Off' %}
{% endGroup %}
```

On

Off

Variants

Set `variant` to `flat`, `3d`, `neon`, or `dot` for different looks.

```
{% group %}
{% led variant='flat' color='blue' %}
{% led variant='3d' color='blue' %}
{% led variant='neon' color='blue' %}
{% led variant='dot' color='blue' %}
{% endGroup %}
```

Shapes and sizes

`shape` (`circle`, `square`, `rectangle`) and `size` (`xs`–`xl`) control the form factor.

```
{% group %}
{% led shape='circle' size='xl' color='grape' %}
{% led shape='square' size='lg' color='grape' %}
{% led shape='rectangle' size='md' color='grape' %}
{% endGroup %}
```

Animation

Turn on `animate` and pick an `animationType` (`pulse`, `flash`, `breathe`, `blink`, `glow`) for an attention-drawing LED.

```
{% group %}
{% led animate animationType='pulse' color='red' label='Pulse' %}
{% led animate animationType='blink' color='yellow' label='Blink' %}
{% led animate animationType='glow' color='cyan' label='Glow' %}
{% endGroup %}
```

Pulse

Blink

Glow

Label from the body

The block body becomes the LED's label when you don't set a plain-text `label`.

```
{% led color='green' labelPosition='right' %}Service healthy{% endLed %}
```

Service healthy

With other components

LEDs read well inside a status card, one row per service.

```
{% card title="System status" %}
{% stack gap='xs' %}
{% led value=true color='green' label='API' labelPosition='right' %}
{% led value=true color='green' label='Database' labelPosition='right' %}
{% led value=false offColor='red' animate animationType='blink' label='Cache'
labelPosition='right' %}
{% endStack %}
{% endCard %}
```

System status

API

Database

Cache

Attributes

Omit any attribute to take its default. Bare flags (e.g. `animate`) become `=True`.

Attribute	Valid values	Description
`value`	`true` / `false`	On/off state. Omit for the package default; `value=false` is an explicitly-off LED.
`color`	Any theme color or CSS color	Color when lit.
`offColor`	Any theme color or CSS color	Color when off.
`size`	`xs`–`xl`	LED size.
`variant`	`flat`, `3d`, `neon`, `dot`	Visual style.
`shape`	`circle`, `square`, `rectangle`	Form factor.
`radius`	`xs`–`xl`	Corner rounding (for square/rectangle).
`animate`	`true` / `false` (default `false`)	Enable animation.
`animationType`	`pulse`, `flash`, `breathe`, `blink`, `glow`	Animation effect.
`animationDuration`	Number (seconds)	Animation speed.
`intensity`	Number	Glow strength.
`label`	String	Descriptive text next to the LED.
`labelPosition`	`left`, `right`	Label placement.
`tooltip`	String	Hover tooltip text.
`gradientFrom`	Any theme color or CSS color	Gradient start color.

`gradientTo`	Any theme color or CSS color	Gradient end color.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

The block body (between `{% led %}` and `{% endLed %}`) is used as the label when no plain-text `label` is set.

CSS Selector

The LED exposes Mantine Styles API selectors you can target with `classNames` in your own React, or style directly:

Selector	Element
`root`	The wrapper element.
`led`	The LED indicator itself.
`label`	The label text area.
`glow`	The outer glow layer.
`light`	The inner light reflection.

Forward a class with `attr={…}` (see below) to scope a stylesheet rule to one instance.

Injecting Attributes

The `attr={…}` channel forwards raw HTML attributes straight onto the rendered LED root — use it for `id`, `class`, `data-*`, ARIA attributes, or anything else not exposed as a typed parameter.

Source: Markdown

```
{% led color='green' attr={'class': 'status-led', 'data-service': 'api'} %}
```

Source: Python

```
component('aardvark', 'led', color='green',
          attr={'class': 'status-led', 'data-service': 'api'})
```

Lightbox

A **full-screen image lightbox**: the tag renders a thumbnail grid, and clicking a thumbnail opens that image full-screen with carousel navigation between the rest. `images` is a JSON array of `{ "src", "alt" }` objects; `cols` sets how many thumbnails sit per row.

A **Community Component** — wraps Lightbox by **rilrom**, **MIT** licensed, npm `@mantine-bites/lightbox` (built on Embla Carousel).

Use it as `{% lightbox %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'lightbox', …)`.

Demonstrations

A thumbnail grid

Pass `images` as a JSON array of `{ "src", "alt" }` objects. Each renders as a thumbnail; click one to open the full-screen lightbox and swipe (or arrow) through the set.

Source: Markdown

```
{% lightbox images='[
  {"src": "https://picsum.photos/id/10/1200/800", "alt": "Forest"},
  {"src": "https://picsum.photos/id/20/1200/800", "alt": "Desk"},
  {"src": "https://picsum.photos/id/30/1200/800", "alt": "Coffee"}
]' %}
```

Source: Python

```
component('aardvark', 'lightbox', images='''[
  {"src": "https://picsum.photos/id/10/1200/800", "alt": "Forest"},
  {"src": "https://picsum.photos/id/20/1200/800", "alt": "Desk"},
  {"src": "https://picsum.photos/id/30/1200/800", "alt": "Coffee"}
]''')
```

Grid width

Set `cols` to change how many thumbnails sit per row — here four across a tighter gallery.

Source: Markdown

```
{% lightbox cols=4 images='[
  {"src": "https://picsum.photos/id/40/1200/800", "alt": "One"},
  {"src": "https://picsum.photos/id/50/1200/800", "alt": "Two"},
  {"src": "https://picsum.photos/id/60/1200/800", "alt": "Three"},
  {"src": "https://picsum.photos/id/70/1200/800", "alt": "Four"}
]' %}
```

Source: Python

```
component('aardvark', 'lightbox', cols=4, images='''[
  {"src": "https://picsum.photos/id/40/1200/800", "alt": "One"},
  {"src": "https://picsum.photos/id/50/1200/800", "alt": "Two"},
  {"src": "https://picsum.photos/id/60/1200/800", "alt": "Three"},
  {"src": "https://picsum.photos/id/70/1200/800", "alt": "Four"}
]''')
```

With other components

Because `images` is plain data, a Python caller can build the gallery from a loop — for example mapping a list of asset URLs — and render it under a Title heading:

Gallery

Source: Markdown

```
{% title order=4 %}Gallery{% endTitle %}

{% lightbox images='[
  {"src": "https://picsum.photos/id/80/1200/800", "alt": "A"},
  {"src": "https://picsum.photos/id/90/1200/800", "alt": "B"}
]' %}
```

Source: Python

```
import json

assets = [
    ('https://picsum.photos/id/80/1200/800', 'A'),
    ('https://picsum.photos/id/90/1200/800', 'B'),
]
images = json.dumps([{'src': src, 'alt': alt} for src, alt in assets])

component('aardvark', 'title', order=4, children='Gallery')
component('aardvark', 'lightbox', images=images)
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description

`images`	JSON array string	The gallery images. Each is a `{ "src", "alt" }` object. Relative paths and `http(s)://` URLs are allowed; unsafe schemes are rejected at build time.
`cols`	integer	Thumbnails per row in the grid. Defaults to `3`.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the grid wrapper element. See Injecting Attributes.

CSS Selector

The grid wrapper carries `data-aardvark-lightbox`, and each thumbnail button carries the class `aardvark-lightbox-thumb`; target them to restyle the gallery:

```
[data-aardvark-lightbox] .aardvark-lightbox-thumb:hover {
  transform: scale(1.05);
}
```

The full-screen overlay and carousel chrome are styled by the upstream `@mantine-bites/lightbox` stylesheet, which the component pulls in automatically.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes onto the grid wrapper element — useful for `id`, `data-*`, ARIA attributes, or anything not exposed as a named param:

Source: Markdown

```
{% lightbox attr={'id': 'demo-gallery', 'data-role': 'lightbox'} images='[{"src":
"https://picsum.photos/id/100/1200/800", "alt": "Sample"}]' %}
```

Source: Python

```
component('aardvark', 'lightbox', attr={'id': 'demo-gallery', 'data-role': 'lightbox'},
          images='[{"src": "https://picsum.photos/id/100/1200/800", "alt": "Sample"}]')
```

Select stepper

A built-in tag for an **option-cycling stepper** — prev/next arrows that step through a list of choices, showing one at a time. Pass `data` as a JSON array (either of plain strings or of `{value, label}` objects) and optionally set a `label`, a starting `defaultValue`, the `orientation`, and `loop` for infinite cycling.

A **Community Component** — wraps SelectStepper by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-select-stepper`.

Demonstrations

Basic stepper

Pass `data` as a JSON array of strings. The stepper starts on the first option; the arrows move through the list.

```
{% selectstepper data='["React","Vue","Svelte","Angular"]' label='Framework' %}
```

Framework

Object data with labels

Use `{value, label}` objects when the stored value differs from what's shown. Set `defaultValue` to start on a specific entry.

```
{% selectstepper
data='[{"value":"sm","label":"Small"},{"value":"md","label":"Medium"},{"value":"lg","label":
"Large"}]' label='Size' defaultValue='md' %}
```

Size

Vertical orientation and looping

`orientation='vertical'` stacks the arrows top/bottom; `loop=true` wraps around past the ends.

```
{% selectstepper data='["Mon","Tue","Wed","Thu","Fri"]' label='Day' orientation='vertical'
loop=true %}
```

Day

Size and description

`size` scales the control; `description` adds helper text below the label.

```
{% selectstepper data='["Low","Medium","High","Critical"]' label='Priority' description='Use
arrows to cycle' size='lg' %}
```

Priority

With other components

Pair a stepper with a Card to build a self-contained picker.

```
{% card title="Plan" %}
{% selectstepper data='["Free","Pro","Team","Enterprise"]' label='Tier' defaultValue='Pro'
%}
{% endCard %}
```

Plan

Tier

Attributes

Omit any attribute to take its default. Bare flags (e.g. `loop`) become `=True`.

Attribute	Valid values	Description
`data`	**Required.** A JSON array of strings **or** `{value, label}` objects	The options to cycle through.
`defaultValue`	String	The value to start on (defaults to the first option).
`label`	String	Label text shown above the control.
`description`	String	Helper text shown below the label.
`orientation`	`horizontal` (default) / `vertical`	Direction of the prev/next arrows.
`size`	`xs` / `sm` / `md` / `lg` / `xl`	Overall control size.
`variant`	String	Visual style variant (passed through to the vendor component).
`loop`	`true` / `false` (default `false`)	Wrap around past the first/last option.
`animate`	`true` / `false` (default `false`)	Animate transitions between options.
`disabled`	`true` / `false` (default `false`)	Disable the whole control.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The component renders inside an island wrapper carrying `data-aardvark-island="SelectStepper"`, with the vendor's own class names on the control inside. Target the wrapper to constrain or align it:

```
[data-aardvark-island="SelectStepper"] {
  max-width: 16rem;
}
```

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes (ids, `data-*`, ARIA) straight onto the rendered root — handy for testing hooks or scripting.

Step

Source: Markdown

```
{% selectstepper data='["A","B","C"]' label='Step' attr={'data-testid': 'grade-stepper'} %}
```

Source: Python

```
component('aardvark', 'selectstepper', data='["A","B","C"]', label='Step',
attr={'data-testid': 'grade-stepper'})
```

Window

`{% window %}` is a **draggable, resizable window** with a title bar and window controls (close, collapse, and a tools menu). Grab the header to move it, drag any edge or corner to resize it, and collapse it to just the title bar. The block body is the window content.

A **Community Component** — wraps Window by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-window`.

This is a different component from the Mantine-core `floatingwindow` builtin, which is a trigger-opened titled card. By default this window sits **inside the page flow** (in a relative host) so the docs stay readable; pass `withinPortal=true` to float it over the whole viewport instead.

Use it as `{% window %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'window', …)`.

Demonstrations

A basic window

The block body is the window content; `title` fills the header. Drag the header to move it and any edge or corner to resize it.

```
{% window title='Readme' %}
Drag me by the header, or grab an edge to resize.

The body renders as **Markdown**, so you can use the usual formatting here.
{% endWindow %}
```

Readme

Drag me by the header, or grab an edge to resize.

The body renders as **Markdown**, so you can use the usual formatting here.

Constrained drag and resize

`draggable` limits how the window moves (`none`, `window`, `header`, or `both`) and `resizable` limits how it resizes (`none`, `vertical`, `horizontal`, or `both`). Here it can only be moved by its header and only resized horizontally.

```
{% window title='Header-drag, horizontal-resize' draggable='header' resizable='horizontal'
%}
This window only moves by its title bar and only grows sideways.
{% endWindow %}
```

Header-drag, horizontal-resize

This window only moves by its title bar and only grows sideways.

Sizing and styling

Seed the initial geometry with `width` and `height`, tint the chrome with `color`, and set the `radius` and `shadow`.

```
{% window title='Styled' color='grape' radius='lg' shadow='xl' width=360 height=220 %}
A larger, rounded, grape-tinted window.
{% endWindow %}
```

Styled

A larger, rounded, grape-tinted window.

Trimming the controls

Turn off the controls you don't need — here only the close button stays, with the collapse and tools buttons removed.

```
{% window title='Just a close button' withCollapseButton=false withToolsButton=false %}
A minimal window chrome.
{% endWindow %}
```

Just a close button

A minimal window chrome.

With other components

The body renders as Markdown, so a window can hold any other component. Here it wraps a `badge` and a `text` block.

```
{% window title='Release notes' width=380 %}
{% badge color='teal' %}v2.0{% endBadge %}

{% text %}A short changelog lives inside the draggable window.{% endText %}
{% endWindow %}
```

Release notes

[v2.0]

A short changelog lives inside the draggable window.

Attributes

Omit any attribute to take its default. Bare flags (e.g. `withinPortal`) become `=true`.

Attribute	Valid values	Description
`title`	text	The header title.

`draggable`	`none`, `window`, `header`, `both` (default `both`)	Where the window can be grabbed to move.
`resizable`	`none`, `vertical`, `horizontal`, `both` (default `both`)	Which directions the window can be resized.
`color`	a Mantine color	Accent color for the chrome.
`radius`	`xs`–`xl` or a number	Corner radius.
`shadow`	`xs`–`xl` (`md` default)	Drop shadow.
`width`	integer px	Initial width (seeds the package's `defaultWidth`).
`height`	integer px	Initial height (seeds the package's `defaultHeight`).
`withinPortal`	`true` / `false` (default)	Float over the whole viewport instead of sitting in the page flow.
`withCloseButton`	`true` (default) / `false`	Show the close button.
`withCollapseButton`	`true` (default) / `false`	Show the collapse button.
`withToolsButton`	`true` (default) / `false`	Show the tools-menu button.
`withBorder`	`true` (default) / `false`	Draw a border around the window.
`attr={…}`	an object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The window renders inside an island wrapper you can target in custom CSS:

```
[data-aardvark-island="Window"] {
  /* your overrides */
}
```

The package also exposes CSS variables on the window root — `--window-background`, `--window-radius`, and `--window-shadow` — which you can set through the `attr` style passthrough below.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (data attributes, ARIA, inline `style`) straight onto the rendered element — handy for hooks your own CSS or scripts key off, or for setting the package's CSS variables.

Custom background

A window whose background is set through the CSS-variable passthrough.

Source: Markdown

```
{% window title='Custom background' attr={'data-testid': 'demo-window', 'style':
'--window-background: var(--mantine-color-blue-light)'} %}
A window whose background is set through the CSS-variable passthrough.
{% endWindow %}
```

Source: Python

```
component('aardvark', 'window', title='Custom background',
          attr={'data-testid': 'demo-window', 'style': '--window-background:
var(--mantine-color-blue-light)'},
          children='A window whose background is set through the CSS-variable passthrough.')
```

QR code

A built-in tag for a **customizable QR code**. Set `value` to the string you want to encode (a URL, text, a vCard, ...) and optionally style it: `color` and `background`, the `dotStyle` and `cornerStyle`, a center `image` overlay, and the `errorCorrectionLevel`.

A **Community Component** — wraps QrCode by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-qr-code`.

Demonstrations

Basic QR code

The only required attribute is `value` — the data to encode.

```
{% qrcode value='https://aardvark.example' %}
```

Colors

`color` tints the data dots (Mantine theme colors work), and `background` sets the backdrop.

```
{% qrcode value='https://aardvark.example' color='indigo' background='white' size='lg' %}
```

Dot and corner styles

`dotStyle` shapes the data modules and `cornerStyle` the finder patterns — each is `square`, `rounded`, or `dots`.

```
{% qrcode value='https://aardvark.example' dotStyle='rounded' cornerStyle='dots'
color='grape' %}
```

Center logo and error correction

Drop an `image` in the middle. A higher `errorCorrectionLevel` (`L` → `H`) keeps the code scannable even with a logo covering part of it.

```
{% qrcode value='https://aardvark.example' image='/favicon.svg' imageRadius='8px'
errorCorrectionLevel='H' size='lg' %}
```

With other components

Put a QR code in a Card to make a shareable “scan me” tile.

```
{% card title="Scan to open the docs" %}
{% qrcode value='https://aardvark.example' color='dark' %}
{% endCard %}
```

Scan to open the docs

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`value`	**Required.** String	The data to encode (URL, text, …).
`size`	`xs` / `sm` / `md` / `lg` / `xl`	Overall size of the code.
`color`	A Mantine color name or CSS color	Color of the data dots.
`background`	A Mantine color name or CSS color	Background color.
`dotStyle`	`square` / `rounded` / `dots`	Shape of the data modules.
`cornerStyle`	`square` / `rounded` / `dots`	Shape of the corner finder patterns.
`image`	A URL or path	A logo/image overlaid at the center.
`imageRadius`	A CSS length (e.g. `8px`)	Border radius of the overlay image.
`errorCorrectionLevel`	`L` / `M` / `Q` / `H`	Error tolerance (7%–30%); higher survives more occlusion.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The component renders inside an island wrapper carrying `data-aardvark-island="QrCode"`, with the vendor's own class names on the SVG inside. Target the wrapper to size or center it:

```
[data-aardvark-island="QrCode"] {
  display: inline-block;
  margin-inline: auto;
}
```

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes (ids, `data-*`, ARIA) straight onto the rendered root — handy for testing hooks or scripting.

Source: Markdown

```
{% qrcode value='https://aardvark.example' attr={'data-testid': 'docs-qr', 'aria-label': 'QR code linking to the docs'} %}
```

Source: Python

```
component('aardvark', 'qrcode', value='https://aardvark.example', attr={'data-testid': 'docs-qr', 'aria-label': 'QR code linking to the docs'})
```

Spinner

`spinner` is an SVG-based loading spinner with several animation variants, configurable segment shapes, gradient colors, glow, and a determinate progress mode. Drop it in wherever something is loading, or drive `progress` for a percentage-based indicator.

A **Community Component** — wraps Spinner by **gfazioli**, **MIT** licensed, installed from npm as `@gfazioli/mantine-spinner`.

Demonstrations

A bare `{% spinner %}` renders the default animated spinner.

```
{% spinner %}
```

Variants

Set `variant` to one of `fade`, `pulse`, `grow`, or `trail` for different animation styles.

```
{% group %}
{% spinner variant='fade' %}
{% spinner variant='pulse' %}
{% spinner variant='grow' %}
{% spinner variant='trail' %}
{% endGroup %}
```

Segments and shapes

`segments` sets how many spokes the spinner draws, and `segmentShape` (`line`, `dot`, or `arc`) sets their shape.

```
{% group %}
{% spinner segments=8 segmentShape='line' %}
{% spinner segments=12 segmentShape='dot' %}
{% spinner segments=6 segmentShape='arc' %}
{% endGroup %}
```

Color and gradient

Set a `color` (any theme color), or pass `gradientFrom` / `gradientTo` for a two-color gradient sweep.

```
{% group %}
{% spinner color='grape' size='lg' %}
{% spinner gradientFrom='indigo' gradientTo='cyan' size='lg' %}
{% endGroup %}
```

Determinate progress

Pass `progress` (0–100) to turn the spinner into a determinate indicator. The body renders in the center, so you can show the percentage.

```
{% spinner progress=68 size='xl' %}68%{% endSpinner %}
```

68%

Glow

`glow` adds a colored bloom around the spinner.

```
{% spinner color='cyan' glow=8 size='lg' %}
```

With other components

Spinners compose with the other built-in tags. Here one sits inside a card next to loading copy.

```
{% card title="Deploying" %}
{% group %}
{% spinner variant='trail' color='blue' %}
Pushing your site to the edge…
{% endGroup %}
{% endCard %}
```

Deploying

Pushing your site to the edge…

Attributes

Omit any attribute to take its default. Numeric attributes keep a deliberate `0`.

Attribute	Valid values	Description
`variant`	`fade`, `pulse`, `grow`, `trail`	Animation style.
`segments`	Integer	Number of spinner segments (spokes).
`segmentShape`	`line`, `dot`, `arc`	Shape of each segment.
`size`	`xs`–`xl` or a number of px	Overall spinner size.
`thickness`	Number	Stroke thickness of each segment.

`inner`	Number	Inner radius of the spinner.
`color`	Any theme color or CSS color	Spinner color.
`direction`	`clockwise`, `counter-clockwise`	Rotation direction.
`duration`	Number (seconds)	One full animation cycle's duration.
`glow`	Number	Glow / bloom intensity (`0` = none).
`progress`	`0`–`100`	Determinate progress; renders a fixed indicator.
`minOpacity`	Number (0–1)	Lowest opacity a fading segment reaches.
`hueRotate`	`true` / `false` (default `false`)	Cycle segment hues for a rainbow effect.
`gradientFrom`	Any theme color or CSS color	Gradient start color.
`gradientTo`	Any theme color or CSS color	Gradient end color.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

The block body (between `{% spinner %}` and `{% endSpinner %}`) renders in the center of the spinner — handy for a progress percentage.

CSS Selector

The spinner is rendered as an SVG element. Its size is driven by the `--spinner-size` CSS custom property (set from the `size` prop and any responsive `size` object via media queries), so you can override sizing from your own stylesheet:

```
.my-loader svg {
  --spinner-size: 64px;
}
```

Target a specific instance by forwarding a class with `attr={…}` (see below).

Injecting Attributes

The `attr={…}` channel forwards raw HTML attributes straight onto the rendered spinner element — use it for `id`, `class`, `data-*`, ARIA attributes, or anything else not exposed as a typed parameter.

Source: Markdown

```
{% spinner color='cyan' attr={'class': 'my-loader', 'aria-label': 'Loading'} %}
```

Source: Python

```
component('aardvark', 'spinner', color='cyan', attr={'class': 'my-loader', 'aria-label':
'Loading'})
```

SplitPane

`{% splitpane %}` places **two resizable panes** with a **styled, draggable resizer** between them — drag the handle to reapportion the space. Provide the panes as the `first` and `second` text params, or put the first pane in the block body and the second in `second`.

A **Community Component** — wraps Split Pane by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-split-pane`.

This is a different component from the Mantine-core `splitter` builtin; reach for `splitpane` when you want the package's richer, themeable resizer handle (variants, a knob, custom colors).

Use it as `{% splitpane %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'splitpane', …)`.

Demonstrations

Body as the first pane

The block body becomes the first pane; `second` holds the second. Drag the resizer to change the split.

```
{% splitpane second='The second pane takes the remaining space.' %}
This is the first pane (from the block body).
{% endSplitpane %}
```

This is the first pane (from the block body).

Both panes as params

Set `first` and `second` to skip the block body entirely — convenient when both panes are short strings or when you are calling from Python.

```
{% splitpane first='Label column' second='Detail column with the bulk of the content.' %}
```

Stacked (horizontal)

The package splits **side by side** by default (`orientation='vertical'`). Pass `orientation='horizontal'` to **stack** the panes with a horizontal resizer between them.

```
{% splitpane first='Top pane' second='Bottom pane' orientation='horizontal' %}
```

A styled resizer

Style the handle with `variant`, `color`, `size`, and `knobSize`. Here a dashed grape resizer with a larger knob.

```
{% splitpane first='Left' second='Right' variant='dashed' color='grape' size='lg'
```

```
knobSize='xl' %}
```

With other components

The block-body pane renders as Markdown, so the first pane can hold any other component. Here it carries a `badge` heading above a short description.

```
{% splitpane second='The detail pane sits to the right and fills the rest of the row.' %}
{% badge color='grape' %}Overview{% endBadge %}

A short summary lives in the first pane.
{% endSplitpane %}
```

[Overview]

A short summary lives in the first pane.

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`first`	text	First pane content. Omit it to use the block body as the first pane instead.
`second`	text	Second pane content.
`orientation`	`vertical` (default, side by side), `horizontal` (stacked)	How the two panes are laid out.
`initialSize`	a width/height, e.g. `'30%'` or `'240px'`	The first pane's initial size.
`variant`	`default`, `filled`, `outline`, `transparent`, `gradient`, `dotted`, `dashed`	The resizer handle style.
`color`	a Mantine color	The resizer color.
`size`	`xs`–`xl` or a number	The resizer thickness.
`radius`	`xs`–`xl` or a number	The resizer corner radius.
`knobSize`	`xs`–`xl` or a number	The size of the resizer's grab knob.

`attr={…}`	an object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The split renders inside an island wrapper you can target in custom CSS:

```
[data-aardvark-island="SplitPane"] {
  /* your overrides */
}
```

The package also exposes CSS variables on the resizer — `--split-resizer-size`, `--split-resizer-color`, `--split-resizer-knob-size`, and more — which you can set through the `attr` style passthrough below.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (data attributes, ARIA, inline `style`) straight onto the rendered element — handy for hooks your own CSS or scripts key off, or for setting the package's CSS variables.

Source: Markdown

```
{% splitpane first='Left' second='Right' attr={'data-testid': 'demo-split', 'style':
'--split-resizer-size: 8px'} %}
```

Source: Python

```
component('aardvark', 'splitpane', first='Left', second='Right',
          attr={'data-testid': 'demo-split', 'style': '--split-resizer-size: 8px'})
```

Compare

`compare` is an **image-comparison slider**: two images stacked behind a divider you drag, hover, or auto-play to reveal one over the other — the classic "before / after" widget. Give it a `leftImage` and a `rightImage` (URLs) and it builds the comparison for you.

A **Community Component** — wraps Compare by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-compare`.

Use it as `{% compare … %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'compare', …)`.

Demonstrations

Drag (default)

The default `variant='drag'` lets the reader drag the divider across the image to reveal more of one side or the other.

```
{% compare
   leftImage='https://images.unsplash.com/photo-1506744038136-46273834b3fb?w=800'
   rightImage='https://images.unsplash.com/photo-1469474968028-56623f02e42e?w=800'
   leftAlt='Lake' rightAlt='Forest' aspectRatio='16/9' radius='md' %}
```

Hover

Set `variant='hover'` so the divider follows the pointer without a drag — move the mouse over the image and the split tracks the cursor.

```
{% compare variant='hover'
   leftImage='https://images.unsplash.com/photo-1506744038136-46273834b3fb?w=800'
   rightImage='https://images.unsplash.com/photo-1469474968028-56623f02e42e?w=800'
   leftAlt='Lake' rightAlt='Forest' aspectRatio='16/9' radius='md' %}
```

Labels and a horizontal divider

`leftLabel` / `rightLabel` add captions to each side, and `angle=90` rotates the divider to a top/bottom comparison.

```
{% compare angle=90 leftLabel='Before' rightLabel='After'
   leftImage='https://images.unsplash.com/photo-1506744038136-46273834b3fb?w=800'
   rightImage='https://images.unsplash.com/photo-1469474968028-56623f02e42e?w=800'
   leftAlt='Before' rightAlt='After' aspectRatio='16/9' radius='md' %}
```

Auto-play

`autoPlay=true` slides the divider back and forth on its own; `autoPlaySpeed` (1-100) tunes the pace.

```
{% compare autoPlay=true autoPlaySpeed=40
   leftImage='https://images.unsplash.com/photo-1506744038136-46273834b3fb?w=800'
   rightImage='https://images.unsplash.com/photo-1469474968028-56623f02e42e?w=800'
   leftAlt='Lake' rightAlt='Forest' aspectRatio='16/9' radius='md' %}
```

With other components

Drop a Compare inside a Paper surface to frame it with the rest of a section.

```
{% paper withBorder=true p='md' radius='md' %}
{% compare variant='hover'
   leftImage='https://images.unsplash.com/photo-1506744038136-46273834b3fb?w=800'
   rightImage='https://images.unsplash.com/photo-1469474968028-56623f02e42e?w=800'
   leftAlt='Lake' rightAlt='Forest' aspectRatio='16/9' radius='md' %}
{% endPaper %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `autoPlay`) become `=True`.

Attribute	Valid values	Description
`leftImage`	URL (string)	The image shown on the left / first side.
`rightImage`	URL (string)	The image shown on the right / second side.
`leftAlt` / `rightAlt`	string	Alt text for each image.
`leftLabel` / `rightLabel`	string	A caption rendered over each side.
`variant`	`drag` / `hover` / `fixed` (default `drag`)	How the divider is controlled.
`aspectRatio`	CSS ratio, e.g. `16/9` (default `16/9`)	Aspect ratio of the container.
`radius`	`xs`–`xl`, a number, or a CSS length (default `md`)	Corner rounding.
`angle`	`0`–`360` (default `0`)	Divider angle: `0` = vertical, `90` = horizontal.

`position`	`0–100`	Controlled divider position.
`defaultPosition`	`0–100` (default `50`)	Initial divider position (uncontrolled).
`minDragBound` / `maxDragBound`	`0–100`	Constrain how far the divider can travel.
`sliderColor`	a Mantine color	Divider line / handle color.
`sliderWidth`	number (px, default `2`)	Divider line width.
`keyboardStep` / `keyboardShiftStep`	number	Keyboard nudge size (and with Shift).
`autoPlay`	`true` / `false` (default `false`)	Continuously slide back and forth.
`autoPlaySpeed`	`1–100` (default `50`)	Auto-play pace (higher = faster).
`autoPlayEasing`	`linear` / `ease-in` / `ease-out` / `ease-in-out` / `spring`	Auto-play easing.
`disabled`	`true` / `false` (default `false`)	Disable all interaction.
`handleOnly`	`true` / `false` (default `false`)	Only drag from the handle, not the whole line.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The component mounts inside a wrapper carrying `data-aardvark-island="Compare"`, and Mantine's Compare exposes Styles API class names you can target:

```
/* The whole comparison container */
[data-aardvark-island='Compare'] .mantine-Compare-root { }
/* The draggable divider line + handle */
[data-aardvark-island='Compare'] .mantine-Compare-sliderLine { }
[data-aardvark-island='Compare'] .mantine-Compare-sliderButton { }
/* The side labels */
[data-aardvark-island='Compare'] .mantine-Compare-leftLabel { }
[data-aardvark-island='Compare'] .mantine-Compare-rightLabel { }
```

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes (data-*, ARIA, etc.) onto the rendered element — they ride a separate channel from the React props, so they never collide with a component prop.

Source: Markdown

```
{% compare
   leftImage='/img/sample-square.svg' rightImage='/img/sample-landscape.svg'
   attr={'data-analytics': 'hero-compare', 'aria-label': 'Before and after'} %}
```

Source: Python

```
component('aardvark', 'compare', leftImage='/img/sample-square.svg',
rightImage='/img/sample-landscape.svg',
          attr={'data-analytics': 'hero-compare', 'aria-label': 'Before and after'})
```

Mask

`mask` is a **cursor-follow spotlight**: it reveals its content through a soft radial or linear gradient that can track the pointer, so only the area under (or near) the cursor is fully visible while the rest fades away. Reach for it to add a "flashlight" reveal over an image, a hero, or a block of text.

A **Community Component** — wraps Mask by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-mask`.

> This is a different component from MaskInput, the formatted text input. The bare name `mask` was free, so this spotlight keeps the natural name.

Use it as `{% mask … %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'mask', …)`. The block body is the content the spotlight reveals.

Demonstrations

Static radial mask

By default the mask is a static radial spotlight centered on the content. The block body is revealed through it.

```
{% mask variant='radial' maskRadius='200px' %}
{% paper withBorder=true p='xl' radius='md' %}
## Spotlight
This whole panel sits behind a soft radial mask — the center is sharp and the edges fade.
{% endPaper %}
{% endMask %}
```

Spotlight

This whole panel sits behind a soft radial mask — the center is sharp and the edges fade.

Cursor-follow spotlight

Set `withCursorMask=true` so the spotlight follows the pointer. `easing` (0-1) controls how smoothly it trails the cursor.

```
{% mask withCursorMask=true maskRadius='160px' easing=0.15 %}
{% paper withBorder=true p='xl' radius='md' %}
Move your cursor across this panel — the bright spotlight tracks the pointer and the rest
dims behind the mask.
{% endPaper %}
{% endMask %}
```

Move your cursor across this panel — the bright spotlight tracks the pointer and the rest dims behind the mask.

Inverted mask

`invertMask=true` flips the effect — the center is hidden and the surroundings are shown, like a vignette punched out of the middle.

```
{% mask withCursorMask=true invertMask=true maskRadius='140px' %}
{% paper withBorder=true p='xl' radius='md' %}
With the mask inverted, the area under the cursor is the part that fades out.
{% endPaper %}
{% endMask %}
```

With the mask inverted, the area under the cursor is the part that fades out.

Reveal on hover

`activation='hover'` keeps the content fully visible until the pointer enters, then engages the mask — a subtle reveal-on-hover.

```
{% mask activation='hover' withCursorMask=true maskRadius='180px' %}
{% paper withBorder=true p='xl' radius='md' %}
This panel is plain until you hover it — then the spotlight switches on.
{% endPaper %}
{% endMask %}
```

This panel is plain until you hover it — then the spotlight switches on.

With other components

Mask wraps any content, including an Image — the spotlight then reveals the photo as the cursor moves.

```
{% mask withCursorMask=true maskRadius='180px' %}
{% image src='https://images.unsplash.com/photo-1469474968028-56623f02e42e?w=800'
alt='Forest' zoom=false %}
{% endMask %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `withCursorMask`) become `=True`.

Attribute	Valid values	Description
`variant`	`radial` / `linear` (default `radial`)	Mask gradient shape.
`withCursorMask`	`true` / `false` (default `false`)	Make the spotlight follow the cursor.

`activation`	`always` / `hover` / `focus` / `pointer` (default `always`)	When the mask is active.
`maskRadius`	number (px) or a CSS length (default `240`)	Radius of the spotlight.
`maskRadiusX` / `maskRadiusY`	number or CSS length	Override the radius per axis.
`maskOpacity`	`0`–`1` (default `1`)	Opacity of the masked content.
`maskFeather`	`0`–`100`	Edge softness: `0` = hard, `100` = full fade (overrides the stops below).
`maskTransparencyStart` / `maskTransparencyEnd`	percentage	Gradient start / end stops.
`maskX` / `maskY`	percentage (default `50`)	Static spotlight center when not following the cursor.
`maskAngle`	number or CSS angle (default `90`)	Gradient angle when `variant='linear'`.
`easing`	`0`–`1` (default `0.12`)	Cursor-follow smoothing (lower = slower trail).
`animation`	`lerp` / `none` (default `lerp`)	`lerp` eases toward the cursor; `none` snaps instantly.
`invertMask`	`true` / `false` (default `false`)	Hide the center and show the surroundings.
`cursorOffsetX` / `cursorOffsetY`	number (px)	Offset the spotlight from the cursor.
`trackPointerOnDocument`	`true` / `false` (default `false`)	Track the pointer across the whole page.
`clampToBounds`	`true` / `false` (default `false`)	Keep the spotlight inside the container.

`clampPadding`	number (px)	Extra padding when `clampToBounds` is on.
`recenterOnResize` / `recenterOnChildrenChange`	`true` / `false` (default `false`)	Recenter the static mask on layout changes.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The component mounts inside a wrapper carrying `data-aardvark-island="Mask"`, and Mantine's Mask exposes Styles API class names you can target:

```
/* The masked container */
[data-aardvark-island='Mask'] .mantine-Mask-root { }
/* The gradient mask layer itself */
[data-aardvark-island='Mask'] .mantine-Mask-mask { }
```

The mask geometry rides CSS variables (`--mask-radius`, `--mask-opacity`, `--mask-transparency-start`, `--mask-transparency-end`) you can also override.

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes (data-*, ARIA, etc.) onto the rendered element — they ride a separate channel from the React props, so they never collide with a component prop.

The spotlight follows your cursor, and the rendered element carries the injected `data-analytics` and `data-section` attributes.

Source: Markdown

```
{% mask withCursorMask=true attr={'data-analytics': 'spotlight', 'data-section': 'hero'} %}
{% paper withBorder=true p='xl' radius='md' %}
The spotlight follows your cursor, and the rendered element carries the injected
`data-analytics` and `data-section` attributes.
{% endPaper %}
{% endMask %}
```

Source: Python

```
component('aardvark', 'mask', withCursorMask=True,
          attr={'data-analytics': 'spotlight', 'data-section': 'hero'},
          children=component('aardvark', 'paper', withBorder=True, p='xl', radius='md',
                             children='The spotlight follows your cursor, and the rendered
element carries the injected `data-analytics` and `data-section` attributes.'))
```

Book

`book` is an iBooks-style **book** whose pages turn with a draggable page-curl: grab the corner of the right page and drag to flip it forward, or the left page to flip back, and the page beneath shows through the curl as it turns. Hand it a set of pages and it hydrates into a fully interactive island in the browser — no JavaScript to write.

A **Community Component** — wraps Book by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-book`.

Demonstrations

A basic book

Pass the pages through `pages` as a JSON array of `{front, back}` objects — `front` and `back` are the two sides of each physical page. Drag a page corner to turn it.

```
{% book pages='[
  {"front":"Chapter 1","back":"Once upon a time…"},
  {"front":"…there was a docs site","back":"that turned its own pages."},
  {"front":"The End","back":""}
]' %}
```

Sizing the page

`width` and `height` set the page geometry in CSS pixels (the book's play-zone is twice the width, for the two-page spread). A taller, narrower book:

```
{% book width=220 height=320 pages='[
  {"front":"A","back":"B"},
  {"front":"C","back":"D"},
  {"front":"E","back":"F"}
]' %}
```

A bound volume with hard covers

Set `withCover=true` to treat the book as a physical bound volume: the first and last pages turn rigid around the spine (hard covers, no curl) and the closed book is centered, sliding into the two-page spread as the cover opens. Pick the curl `variant` (`flat` or `rounded`) and set `revealBackground` for the inside-cover color.

```
{% book withCover=true variant='rounded' revealBackground='#2b2b3a' pages='[
  {"front":"Front cover","back":"Title page"},
  {"front":"Page 1","back":"Page 2"},
  {"front":"Page 3","back":"Back cover"}
]' %}
```

With other components

Each page's `front`/`back` is rendered content, so a Book sits naturally under an introductory Title:

```
{% title order=3 %}Our story{% endTitle %}
{% book pages='[
  {"front":"2021","back":"We started."},
  {"front":"2023","back":"We shipped."},
  {"front":"Today","back":"You are reading it."}
]' %}
```

Our story

Attributes

Omit any attribute to take its default. Bare flags (e.g. `withCover`) become `=True`.

Attribute	Valid values	Description
`pages`	JSON array of `{front, back, props}`	The pages. `front`/`back` are the two sides; `props` (optional) overrides per-page settings.
`width`	int px (default `300`)	Page width. The play-zone is twice this.
`height`	int px (default `600`)	Page height.
`variant`	`flat` / `rounded`	The curl renderer used for every page.
`revealBackground`	Mantine color / CSS color	Inside-cover background, painted under the page stack.
`defaultPage`	int (default `0`)	Initial face index (`0` is the closed book).
`riffleDuration`	int ms (default `1000`)	Total time budget for a multi-page riffle.
`withCover`	`true` / `false` (default `false`)	Treat as a bound volume: hard covers + centered closed book.
`disabled`	`true` / `false` (default `false`)	Disable the drag interaction on every page.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The island mounts under a stable wrapper you can target from your own CSS:

```
[data-aardvark-island="Book"] {
  /* your overrides */
}
```

The upstream component also exposes Mantine Styles-API class names (`root`, `page`) and CSS variables (`--curl-page-width`, `--curl-page-height`, `--curl-reveal-background`) for finer control.

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes onto the rendered root element — handy for analytics hooks, test ids, or ARIA attributes the component doesn't expose as a prop. The object passes through verbatim (it is not a React prop):

Source: Markdown

```
{% book attr={'data-testid': 'story-book', 'data-analytics': 'about-page'} pages='[
  {"front":"Hi","back":"There"}
]' %}
```

Source: Python

```
component('aardvark', 'book',
          attr={'data-testid': 'story-book', 'data-analytics': 'about-page'},
          pages='''[
  {"front":"Hi","back":"There"}
]''')
```

DepthSelect

`depthselect` is a 3D **stack select** — a "Time Machine"-style column of cards that recede into depth. The front card is in focus; the ones behind shrink, drop, fade, and blur with distance, and you riffle forward and back with the built-in controls, the scroll wheel, or the keyboard. Drop a set of cards into a page with no JavaScript; it hydrates into a fully interactive island in the browser.

A **Community Component** — wraps DepthSelect by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-depth-select`.

Demonstrations

A basic stack

Pass the cards through `data` as a JSON array of `{value, view}` objects: `value` is the unique id and `view` is the card's content. The first card is selected by default.

```
{% depthselect data='[
  {"value":"one","view":"Card One"},
  {"value":"two","view":"Card Two"},
  {"value":"three","view":"Card Three"},
  {"value":"four","view":"Card Four"},
  {"value":"five","view":"Card Five"}
]' %}
```

Tuning the depth

`visibleCards` sets how many cards show in the stack, and `scaleStep`, `translateYStep`, `opacityStep`, and `blurStep` control how sharply each card recedes per depth level. Here a deeper stack with a gentler falloff:

```
{% depthselect visibleCards=6 scaleStep=0.06 translateYStep=20 opacityStep=0.1 blurStep=0
data='[
  {"value":"a","view":"Alpha"},
  {"value":"b","view":"Bravo"},
  {"value":"c","view":"Charlie"},
  {"value":"d","view":"Delta"},
  {"value":"e","view":"Echo"},
  {"value":"f","view":"Foxtrot"}
]' %}
```

Looping, controls, and selection

Set `loop=true` to wrap from the last card back to the first, `controlsPosition` to move the nav buttons, and `defaultValue` to start on a specific card. Turn off `withControls` or `withScrollNavigation` to drop those affordances.

```
{% depthselect loop=true controlsPosition='left' defaultValue='two' data='[
  {"value":"one","view":"First"},
  {"value":"two","view":"Second"},
  {"value":"three","view":"Third"}
]' %}
```

With other components

Each card's `view` is rendered content, so a DepthSelect reads naturally next to other tags — for example introduced by a Title and a Text lead-in:

```
{% title order=3 %}Recent snapshots{% endTitle %}
{% text c='dimmed' %}Riffle through the stack to pick one.{% endText %}
{% depthselect data='[
  {"value":"mon","view":"Monday"},
  {"value":"tue","view":"Tuesday"},
  {"value":"wed","view":"Wednesday"}
]' %}
```

Recent snapshots

Riffle through the stack to pick one.

Attributes

Omit any attribute to take its default. Bare flags (e.g. `loop`) become `=True`.

Attribute	Valid values	Description
`data`	JSON array of `{value, view}`	The cards. `value` is the unique id; `view` is the card content.
`defaultValue`	string	The `value` of the card selected on first render (defaults to the first).
`visibleCards`	int (default `4`)	How many cards are visible in the stack.
`controlsPosition`	`right` (default) / `left`	Where the nav controls sit relative to the stack.
`transitionDuration`	int ms (default `400`)	Duration of the riffle animation.
`scaleStep`	float (default `0.1`)	Scale reduction per depth level.
`translateYStep`	int px (default `30`)	Vertical offset per depth level.

`opacityStep`	float (default `0.15`)	Opacity reduction per depth level.
`blurStep`	int px (default `1`)	Blur increment per depth level.
`ariaLabel`	string (default `Depth select`)	Accessible label for the component.
`withControls`	`true` (default) / `false`	Show the built-in navigation controls.
`withScrollNavigation`	`true` (default) / `false`	Enable mouse-wheel navigation.
`loop`	`true` / `false` (default `false`)	Wrap from the last card to the first and back.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The island mounts under a stable wrapper you can target from your own CSS:

```
[data-aardvark-island="DepthSelect"] {
  /* your overrides */
}
```

The upstream component also exposes Mantine Styles-API class names (`root`, `stack`, `card`, `controls`, `controlUp`, `controlDown`, `controlLabel`) and CSS variables (`--ds-scale-step`, `--ds-translate-y-step`, `--ds-opacity-step`, `--ds-blur-step`, `--ds-visible-cards`, `--ds-transition-duration`) for finer control.

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes onto the rendered root element — handy for analytics hooks, test ids, or ARIA attributes the component doesn't expose as a prop. The object passes through verbatim (it is not a React prop):

Source: Markdown

```
{% depthselect attr={'data-testid': 'snapshot-stack', 'data-analytics': 'gallery'} data='[
  {"value":"one","view":"One"},
  {"value":"two","view":"Two"}
]' %}
```

Source: Python

```
component('aardvark', 'depthselect',
          attr={'data-testid': 'snapshot-stack', 'data-analytics': 'gallery'}, data='''[
  {"value":"one","view":"One"},
  {"value":"two","view":"Two"}
]''')
```

Scene

`scene` is a **decorative animated background**: it layers effects — gradients, glow blobs, dot grids, and noise grain — behind the block body. Turn layers on with the flags `gradient`, `glow`, `dotGrid`, and `noise`, tune each with a few color shortcuts, or pass `layersJson` for full control. The block body is the foreground content rendered over the scene.

A **Community Component** — wraps Scene by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-scene`.

Use it as `{% scene … %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'scene', …)`.

Demonstrations

A gradient backdrop

Turn on the `gradient` layer and set `gradientFrom` / `gradientTo` to wash the background with a color blend behind your content.

```
{% scene gradient gradientFrom='indigo' gradientTo='grape' %}
### Layered over a gradient
A soft color wash sits behind this text.
{% endScene %}
```

Layered over a gradient

A soft color wash sits behind this text.

Glow and a dot grid

Stack layers by combining flags — here a `glow` blob and a `dotGrid`, each with its own color shortcut (`glowColor`, `dotColor`).

```
{% scene glow glowColor='cyan' dotGrid dotColor='blue' %}
### Glow + dots
Two layers compose back-to-front behind the content.
{% endScene %}
```

Glow + dots

Two layers compose back-to-front behind the content.

Mouse-tracking interaction

Add `interactive` so the scene responds to the pointer, and `noise` for a subtle film grain.

```
{% scene interactive glow glowColor='grape' noise %}
```

```
### Move your mouse
The glow follows the pointer; a noise layer adds grain.
{% endScene %}
```

Move your mouse

The glow follows the pointer; a noise layer adds grain.

Full control with `layersJson`

When the flag shortcuts aren't enough, pass `layersJson` — a JSON array of `{kind, …}` descriptors, where `kind` is the layer type and the rest are that layer's props. Order is back-to-front.

```
{% scene
layersJson='[{"kind":"gradient","type":"radial","from":"teal","to":"blue"},{"kind":"glow","c
olor":"teal"},{"kind":"noise"}]' %}
### Composed from JSON
A radial gradient, a glow, and noise — fully specified.
{% endScene %}
```

Composed from JSON

A radial gradient, a glow, and noise — fully specified.

With other components

Use a Scene as the backdrop for a hero, then layer other built-in tags over it — here a `{% title %}` and a `{% badge %}`.

```
{% scene gradient gradientFrom='blue' gradientTo='cyan' glow %}
{% title order=2 %}Build with Aardvark{% endTitle %}
Status: {% badge color='green' %}stable{% endBadge %}
{% endScene %}
```

Build with Aardvark

Status: [stable]

Attributes

Omit any attribute to take its default. Bare flags (e.g. `gradient`, `glow`) become `=True`.

Attribute	Valid values	Description
`gradient`	`true` / `false` (default `false`)	Turn on the gradient layer.

`gradientFrom` / `gradientTo`	a Mantine color	Gradient start / end color.
`gradientType`	`radial` / `linear` / `conic`	Gradient shape.
`glow`	`true` / `false` (default `false`)	Turn on a glow blob layer.
`glowColor`	a Mantine color	Glow color.
`dotGrid`	`true` / `false` (default `false`)	Turn on a dot-grid layer.
`dotColor`	a Mantine color	Dot color.
`noise`	`true` / `false` (default `false`)	Turn on a noise / grain layer.
`layersJson`	a JSON array of `{kind, …}` descriptors	Full layer control; wins over the flags when set.
`fullscreen`	`true` / `false` (default `false`)	Fix the scene to cover the whole viewport.
`zIndex`	number	Stacking order of the scene.
`interactive`	`true` / `false` (default `false`)	Track the pointer (e.g. move the glow).
`interactiveEasing`	number	Easing factor for the interactive tracking.
`reducedMotion`	`auto` / `always` / `never`	Honor reduced-motion preferences.
`lazy`	`true` / `false` (default `false`)	Defer mounting until the scene scrolls into view.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The scene mounts inside a wrapper carrying `data-aardvark-island="Scene"`, and Mantine's Scene exposes Styles API class names you can target:

```
/* The scene container that holds the layers + your content */
[data-aardvark-island='Scene'] .mantine-Scene-root { }
/* The foreground content layer (your block body) */
[data-aardvark-island='Scene'] .mantine-Scene-content { }
/* Individual effect layers (gradient / glow / dot grid / noise) */
[data-aardvark-island='Scene'] .mantine-Scene-layer { }
```

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes (data-*, ARIA, etc.) onto the rendered element — they ride a separate channel from the React props, so they never collide with a component prop.

Foreground content

Source: Markdown

```
{% scene gradient gradientFrom='indigo' gradientTo='grape'
   attr={'data-analytics': 'hero-scene', 'aria-hidden': 'true'} %}
Foreground content
{% endScene %}
```

Source: Python

```
component('aardvark', 'scene', gradient=True, gradientFrom='indigo', gradientTo='grape',
          attr={'data-analytics': 'hero-scene', 'aria-hidden': 'true'}, children='Foreground
content')
```

Flip

`flip` is a two-faced surface that rotates between a **front** and a **back** face. The block body is the front face; set `back` for the back face. By default it flips on click; turn on `swipeable` for touch flips, and use `direction`, `duration`, `easing`, and `perspective` to shape the animation.

A **Community Component** — wraps Flip by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-flip`.

Use it as `{% flip %}…{% endFlip %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'flip', …)`.

Demonstrations

The block body is the front face. Set `back` for the back face, then click the card to flip it. Give it a width and height (`w` / `h`) so both faces share a box.

This is the front face. Click to flip.

```
{% flip w=320 h=160 back='And this is the back. Click again to flip back.' %}
This is the front face. Click to flip.
{% endFlip %}
```

Flip direction

Set `direction` to `vertical` to flip top-over-bottom instead of the default `horizontal`.

Vertical flip — click me.

```
{% flip w=320 h=160 direction='vertical' back='Flipped vertically.' %}
Vertical flip — click me.
{% endFlip %}
```

Easing and duration

`easing` accepts `ease`, `ease-in`, `ease-out`, `ease-in-out` (the default), `linear`, or `spring`; `duration` is the flip time in seconds (default `0.8`), and `perspective` is a CSS length (e.g. `1200px`, default `1000px`) that sets the 3D depth.

A bouncy spring flip — click me.

```
{% flip w=320 h=160 easing='spring' duration=0.9 perspective='1200px' back='Sproing!' %}
A bouncy spring flip — click me.
{% endFlip %}
```

Starting flipped

Set `defaultFlipped=true` to show the back face first.

The hidden front face.

```
{% flip w=320 h=160 defaultFlipped=true back='You are looking at the back face first.' %}
The hidden front face.
{% endFlip %}
```

With other components

The front face is the block body, so it can hold any other component. Here a Card is the front face and a short message is the back.

Flip me

This whole card is the front face of a flip — click anywhere on it.

```
{% flip w=340 h=200 back='Thanks for flipping! The back face is plain text.' %}
{% card title='Flip me' withBorder=true %}This whole card is the front face of a flip —
click anywhere on it.{% endCard %}
{% endFlip %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `swipeable`) become `=True`.

Attribute	Valid values	Description
`back`	Text	The back face content (plain text; HTML-escaped).
`flipped`	`true` / `false`	Controlled flip state.
`defaultFlipped`	`true` / `false` (default `false`)	Show the back face first.
`direction`	`horizontal` (default) / `vertical`	Axis the card rotates around.
`easing`	`ease` / `ease-in` / `ease-out` / `ease-in-out` (default) / `linear` / `spring`	Animation easing.

`duration`	A number (seconds, default `0.8`)	Flip animation duration.
`perspective`	A number	3D perspective depth.
`disabled`	`true` / `false` (default `false`)	Freeze the flip.
`lazyBack`	`true` / `false` (default `false`)	Render the back face only once it's first shown.
`swipeable`	`true` / `false` (default `false`)	Flip on touch swipe.
`swipeThreshold`	A number (default `50`)	Swipe distance needed to flip.
`w`, `h`	A number (px) or Mantine size token	Width / height of the flip box.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The package exposes these hooks you can target from your site CSS (e.g. a project stylesheet):

Selector	Element
`.flip-container`	The container that holds both faces.
`.flip-front-face`	The front face.
`.flip-back-face`	The back face.
`[data-aardvark-flip]`	The Aardvark island wrapper around the Flip root.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (id, class, data-*, ARIA, inline style) onto the rendered element — an escape hatch for anything not covered by a named attribute above.

Front face

Source: Markdown

```
{% flip w=320 h=160 back='Back face' attr={'id': 'hero-flip', 'data-analytics': 'flip'} %}
Front face
```

```
{% endFlip %}
```

Source: Python

```
component('aardvark', 'flip', w=320, h=160, back='Back face', children='Front face',
          attr={'id': 'hero-flip', 'data-analytics': 'flip'})
```

Parallax

`parallax` tilts the block body in **3D toward the pointer** for an interactive parallax effect — hover (or, on touch, drag) over it and it leans. Tune the motion with `maxRotation`, `perspective`, `hoverScale`, and `transitionDuration`, and layer on `lightEffect`, `glareEffect`, and `shadowEffect` for shine and depth.

A **Community Component** — wraps Parallax by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-parallax`.

Use it as `{% parallax %}…{% endParallax %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'parallax', …)`.

Demonstrations

The block body is the content to tilt. Give it a width and height (`w` / `h`) and hover over it to see the effect.

Hover me

This card tilts in 3D toward your pointer.

```
{% parallax w=320 h=180 %}
{% card title='Hover me' withBorder=true %}This card tilts in 3D toward your pointer.{%
endCard %}
{% endParallax %}
```

Tilt amount and perspective

`maxRotation` caps how far it leans (degrees); `perspective` sets the 3D depth; `hoverScale` zooms slightly on hover.

Steep tilt

A stronger lean with a slight zoom on hover.

```
{% parallax w=320 h=180 maxRotation=25 perspective=600 hoverScale=1.05 %}
{% card title='Steep tilt' withBorder=true %}A stronger lean with a slight zoom on hover.{%
endCard %}
{% endParallax %}
```

Light and glare

Turn on `lightEffect` for a moving highlight and `glareEffect` for a glossy sheen that follows the pointer.

Shiny

A light highlight and a glare sweep track the pointer.

```
{% parallax w=320 h=180 lightEffect glareEffect glareMaxOpacity=0.4 %}
{% card title='Shiny' withBorder=true %}A light highlight and a glare sweep track the
pointer.{% endCard %}
{% endParallax %}
```

Spring and shadow

`springEffect` gives the tilt a bouncy spring (tuned by `springStiffness` / `springDamping`), and `shadowEffect` adds a dynamic drop shadow.

Bouncy

A springy tilt with a moving shadow underneath.

```
{% parallax w=320 h=180 springEffect shadowEffect shadowColor='black' shadowBlur=24 %}
{% card title='Bouncy' withBorder=true %}A springy tilt with a moving shadow underneath.{%
endCard %}
{% endParallax %}
```

With other components

Parallax wraps any content — a Card, an image, a heading. Here it tilts a title and a line of text.

Parallax

Hover anywhere on this panel to tilt it in 3D.

```
{% parallax w=340 h=160 lightEffect %}
{% title order=3 %}Parallax{% endTitle %}
Hover anywhere on this panel to tilt it in 3D.
{% endParallax %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `lightEffect`) become `=True`.

Attribute	Valid values	Description
`disabled`	`true` / `false` (default `false`)	Freeze the tilt.
`perspective`	A number	3D perspective depth.
`maxRotation`	A number (degrees)	Maximum tilt angle.
`hoverScale`	A number (default `1`)	Scale applied on hover.
`threshold`	A number	Pointer-movement threshold before tilting.
`transitionDuration`	A number (ms, default `300`)	Tilt transition time.
`transitionEasing`	A CSS easing (default `ease-out`)	Tilt transition easing.
`touchEnabled`	`true` / `false` (default on)	Respond to touch drags.
`resetOnLeave`	`true` / `false` (default on)	Return to flat when the pointer leaves.
`invertRotation`	`true` / `false` (default `false`)	Flip the tilt direction.
`lightEffect`	`true` / `false` (default `false`)	Pointer-following highlight.
`lightColor`	A CSS color	Highlight color.
`lightIntensity`	A number	Highlight strength.
`glareEffect`	`true` / `false` (default `false`)	Glossy glare sweep.
`glareColor`	A CSS color	Glare color.
`glareMaxOpacity`	A number `0`–`1`	Maximum glare opacity.

`shadowEffect`	`true` / `false` (default `false`)	Dynamic drop shadow.
`shadowColor`	A CSS color	Shadow color.
`shadowBlur`	A number	Shadow blur.
`springEffect`	`true` / `false` (default `false`)	Bouncy spring physics.
`springStiffness`	A number (default `150`)	Spring stiffness.
`springDamping`	A number (default `12`)	Spring damping.
`gyroscopeEnabled`	`true` / `false` (default `false`)	Tilt by device gyroscope.
`keyboardEnabled`	`true` / `false` (default `false`)	Tilt by arrow keys.
`w`, `h`	A number (px) or Mantine size token	Width / height of the panel.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The parallax effect is owned by the package's stylesheet (loaded automatically). Aardvark adds one stable hook for site overrides:

Selector	Element
`[data-aardvark-parallax]`	The Aardvark island wrapper around the Parallax root.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (id, class, data-*, ARIA, inline style) onto the rendered element — an escape hatch for anything not covered by a named attribute above.

Hover me

Content

Source: Markdown

```
{% parallax w=320 h=180 lightEffect attr={'id': 'hero-parallax', 'data-analytics': 'tilt'}
%}
{% card title='Hover me' withBorder=true %}Content{% endCard %}
{% endParallax %}
```

Source: Python

```
component('aardvark', 'parallax', w=320, h=180, lightEffect=True,
          attr={'id': 'hero-parallax', 'data-analytics': 'tilt'},
          children=component('aardvark', 'card', title='Hover me', withBorder=True,
children='Content'))
```

Reflection

`reflection` paints a **mirrored reflection** beneath the block body — the glassy floor-reflection look. Turn on `ripple` for an animated water ripple, tune the mirror with `reflectionOpacity` / `reflectionDistance` / `reflectionBlur`, and add a contact shadow with the `shadow*` props.

A **Community Component** — wraps Reflection by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-reflection`.

Use it as `{% reflection %}…{% endReflection %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'reflection', …)`.

Demonstrations

The block body is the content to reflect. With no props you get a clean static mirror beneath it.

Reflected

A mirrored copy of this card is painted below it.

```
{% reflection %}
{% card title='Reflected' withBorder=true w=260 %}A mirrored copy of this card is painted
below it.{% endCard %}
{% endReflection %}
```

Water ripple

Turn on `ripple` for an animated water surface, then shape it with `rippleStrength`, `rippleFrequency`, and `rippleSpeed`.

On the water

The reflection below ripples like water.

```
{% reflection ripple rippleStrength=8 rippleSpeed=3 %}
{% card title='On the water' withBorder=true w=260 %}The reflection below ripples like
water.{% endCard %}
{% endReflection %}
```

Mirror opacity and distance

`reflectionOpacity` fades the mirror, `reflectionDistance` pushes it away from the source, and `reflectionBlur` softens it.

Soft mirror

A fainter, slightly blurred reflection set a little lower.

```
{% reflection reflectionOpacity=0.5 reflectionDistance=10 reflectionBlur=2 %}
{% card title='Soft mirror' withBorder=true w=260 %}A fainter, slightly blurred reflection
set a little lower.{% endCard %}
{% endReflection %}
```

Contact shadow

Add a grounding shadow under the content with `shadowSize`, `shadowOpacity`, `shadowBlur`, and `shadowColor`.

Grounded

A soft contact shadow sits beneath this card.

```
{% reflection shadowSize=24 shadowOpacity=0.3 shadowBlur=12 shadowColor='black' %}
{% card title='Grounded' withBorder=true w=260 %}A soft contact shadow sits beneath this
card.{% endCard %}
{% endReflection %}
```

With other components

Reflection wraps anything — an image, a Card, a Badge. Here it mirrors a badge group.

[New][Beta]

```
{% reflection ripple reflectionOpacity=0.6 %}
{% group %}{% badge variant='filled' color='grape' %}New{% endBadge %}{% badge
variant='light' %}Beta{% endBadge %}{% endGroup %}
{% endReflection %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `ripple`) become `=True`.

Attribute	Valid values	Description
`ripple`	`true` / `false` (default `false`)	Animate the reflection as a water surface.
`rippleStrength`	A number (default `4`)	Ripple amplitude.
`rippleFrequency`	A number (default `0.01`)	Ripple frequency.
`rippleSpeed`	A number (default `3`)	Ripple animation speed.
`rippleOctaves`	A number (default `1`)	Layers of ripple noise.
`rippleSeed`	A number (default `1`)	Ripple random seed.
`reflectionOpacity`	A number `0`–`1`	Opacity of the mirrored copy.
`reflectionDistance`	A number	Gap between the content and its reflection.
`reflectionBlur`	A number	Blur applied to the reflection.
`reflectionStretch`	A number	Vertical stretch of the reflection.
`shadowSize`	A number	Size of the contact shadow.
`shadowOffset`	A number	Shadow vertical offset.
`shadowOpacity`	A number `0`–`1`	Shadow opacity.
`shadowBlur`	A number	Shadow blur.
`shadowColor`	A CSS color	Shadow color.
`disableChildren`	`true` / `false` (default `false`)	Render only the reflection, not the source content.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The reflection effect is owned by the package's stylesheet (loaded automatically). Aardvark adds one stable hook for site overrides:

Selector	Element
`[data-aardvark-reflection]`	The Aardvark island wrapper around the Reflection root.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (id, class, data-*, ARIA, inline style) onto the rendered element — an escape hatch for anything not covered by a named attribute above.

Reflected

Content

Source: Markdown

```
{% reflection ripple attr={'id': 'hero-reflection', 'data-analytics': 'reflect'} %}
{% card title='Reflected' withBorder=true %}Content{% endCard %}
{% endReflection %}
```

Source: Python

```
component('aardvark', 'reflection', ripple=True, attr={'id': 'hero-reflection',
'data-analytics': 'reflect'},
          children=component('aardvark', 'card', title='Reflected', withBorder=True,
children='Content'))
```

JsonTree

`jsontree` renders any JSON-serializable value — an object, array, or scalar — as an **interactive, collapsible tree** with syntax highlighting. Readers can expand and collapse nodes, copy individual values or the whole document, search keys and values, and read the full JSON path on hover. It hydrates into a fully interactive island in the browser, with no JavaScript to write.

You pass the value to display as a JSON **string** through `data`; the tag parses it at build time and hands the parsed value to the viewer. Use it as `{% jsontree %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'jsontree', …)`.

A **Community Component** — wraps JsonTree by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-json-tree`.

Demonstrations

Basic tree

Pass a JSON object string through `data`. By default the first couple of levels auto-expand:

```
{% jsontree
data='{"name":"Aardvark","version":"1.0","tags":["docs","static"],"meta":{"stars":42,"active
":true}}' %}
```

Line numbers, indent guides, and item counts

Turn on `showLineNumbers`, `showIndentGuides`, and `showItemsCount` for a denser, more inspectable layout — handy for larger payloads:

```
{% jsontree
data='{"id":7,"items":[{"sku":"A1","qty":2},{"sku":"B2","qty":5}],"shipped":false}'
showLineNumbers=true showIndentGuides=true showItemsCount=true %}
```

Search, copy, and a bordered container

`withSearch` adds a filter box that highlights and auto-expands matches; `withCopyAll` adds a global copy button; `withBorder` wraps the tree in a bordered Paper. `rootName` labels the root node:

```
{% jsontree
data='{"user":{"name":"Ada","roles":["admin","editor"]},"flags":{"beta":true,"darkMode":fals
e}}' rootName="config" withSearch=true withCopyAll=true withCopyToClipboard=true
withBorder=true %}
```

Controlling expansion depth

`maxDepth` sets how many levels auto-expand: `0` collapses everything, a positive number expands that many levels, and `-1` expands the whole tree. `withExpandAll` adds an expand/collapse-all control:

```
{% jsontree data='{"a":{"b":{"c":{"d":"deep"}}},"list":[1,2,3]}' maxDepth=1
withExpandAll=true %}
```

With other components

A JSON tree sits naturally inside a card alongside other content — here a short text lead-in above an API-response viewer:

Example API response

```
{% card %}
{% text size='sm' c='dimmed' %}Example API response{% endText %}

{% jsontree
data='{"status":200,"data":{"id":"u_123","email":"ada@example.com"},"cached":false}'
rootName="response" withCopyAll=true %}
{% endCard %}
```

Attributes

Every attribute is optional; omit one to take its upstream default. The body is ignored — `jsontree` is a viewer, not a container.

Attribute	Valid values	Description
`data`	JSON string	The value to display — an object, array, or scalar. Parsed at build time; a malformed value warns and degrades to an empty object.
`rootName`	String	Label for the root node (default `root`).
`maxDepth`	Integer	Levels to auto-expand: `0` collapsed, `-1` expand all (default `2`).

`size`	`xs`, `sm`, `md`, `lg`, `xl`, or a number	Font scale of the tree.
`maxHeight`	CSS length (string)	Cap the height; the tree scrolls past it.
`displayFunctions`	`as-string`, `hide`, `as-object`	How function values are rendered (default `as-string`).
`borderRadius`	`xs`, `sm`, `md`, `lg`, `xl`	Paper radius when `withBorder` is on (default `sm`).
`searchPlaceholder`	String	Placeholder for the search box when `withSearch` is on.
`defaultExpanded`	`true` / `false` (default `false`)	Expand all nodes by default.
`withBorder`	`true` / `false` (default `false`)	Wrap the tree in a bordered Paper.
`showLineNumbers`	`true` / `false` (default `false`)	Show line numbers down the side.
`showIndentGuides`	`true` / `false` (default `false`)	Show vertical indent guides.
`showItemsCount`	`true` / `false` (default `false`)	Show item counts for objects and arrays.
`showPathOnHover`	`true` / `false` (default `false`)	Show the full JSON path in a tooltip on hover.
`withCopyToClipboard`	`true` / `false` (default `false`)	Show a copy button on each node.

`withCopyAll`	`true` / `false` (default `false`)	Show a global copy-all button in the toolbar.
`withExpandAll`	`true` / `false` (default `false`)	Show the expand/collapse-all control.
`withSearch`	`true` / `false` (default `false`)	Show the search toggle and filter box.
`withKeyCountBadge`	`true` / `false` (default `false`)	Show a badge with the total key/item count.
`stickyHeader`	`true` / `false` (default `false`)	Keep the toolbar header sticky while scrolling.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The viewer is an island wrapper carrying `data-aardvark-island="JsonTree"`; the upstream component owns the inner tree chrome. Style or target it from your own CSS through the island attribute or any `attr`-supplied id/class:

```
/* The JsonTree island wrapper */
[data-aardvark-island="JsonTree"] {
  margin-block: var(--mantine-spacing-md);
}
```

The upstream package also exposes a rich set of CSS variables (e.g. `--json-tree-color-string`, `--json-tree-color-number`, `--json-tree-color-key`, `--json-tree-indent-guide-color-0`) and Styles API selectors — see the upstream documentation for the full list.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (an `id`, `class`, `data-*`, ARIA attributes, inline event handlers) straight onto the rendered element — exactly like `component('aardvark', 'jsontree', attr={…})`. These ride the `data-aardvark-attr` channel, not React props, so they don't collide with the component's own props:

Source: Markdown

```
{% jsontree data='{"hello":"world"}' attr={'id': 'demo-tree', 'data-role': 'json-viewer'} %}
```

Source: Python

```
component('aardvark', 'jsontree', data='{"hello":"world"}',
          attr={'id': 'demo-tree', 'data-role': 'json-viewer'})
```

LensSelect

`lensselect` is a **macOS Dock-style** select: a row (or column) of items where the one under your cursor swells and its neighbours scale by proximity, giving that fisheye-lens magnification you know from the Dock. Hand it a list of items, or a numeric range, and it hydrates into a fully interactive island in the browser — no JavaScript to write.

A **Community Component** — wraps LensSelect by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-lens-select`.

Demonstrations

A row of items

Pass items through `data` as a JSON array of `{value, view}` — `view` is the item's content and is optional (omit it for the default pill). Hover across the row to see the lens follow your cursor.

```
{% lensselect data='[
  {"value":"home","view":"⌂"},
  {"value":"search","view":"⌕"},
  {"value":"mail","view":"✉"},
  {"value":"music","view":"♫"},
  {"value":"photos","view":"✿"},
  {"value":"settings","view":"⚙"}
]' %}
```

A generated numeric range

With no `data`, give `count` for values `1..N`, or `min`/`max`/`step` for a custom range — LensSelect renders default pills for each. This makes a compact, scrubber-like picker:

```
{% lensselect min=0 max=100 step=10 %}
```

macOS Dock feel

Turn on `expandOnHover` so items push their neighbours apart as they magnify (the real Dock behaviour), raise `magnification`, and widen `lensRange` so more neighbours respond:

```
{% lensselect expandOnHover=true magnification=2.5 lensRange=4 itemSize='40' gap='8' data='[
  {"value":"a","view":"A"},
  {"value":"b","view":"B"},
  {"value":"c","view":"C"},
  {"value":"d","view":"D"},
  {"value":"e","view":"E"}
]' %}
```

Vertical orientation and a hover selection mode

Set `orientation='vertical'` for a column, `selectionMode='hover'` to auto-select on hover (no click needed), and `withIndicator=true` to show a marker beside the active item:

```
{% lensselect orientation='vertical' selectionMode='hover' withIndicator=true data='[
  {"value":"one","view":"1"},
  {"value":"two","view":"2"},
  {"value":"three","view":"3"},
  {"value":"four","view":"4"}
]' %}
```

With other components

A LensSelect makes a tidy inline control, so it sits well beside a label rendered with Text:

```
{% text fw=600 %}Rate it{% endText %}
{% lensselect count=5 withScale=true magnification=2 %}
```

Rate it

Attributes

Omit any attribute to take its default. Bare flags (e.g. `expandOnHover`) become `=True`.

Attribute	Valid values	Description
`data`	JSON array of `{value, view}`	The items. `value` is the unique id; `view` (optional) is the content.
`defaultValue`	string	The `value` selected on first render.
`count`	int	Generate values `1..count` when no `data` is given.
`min` / `max` / `step`	float	Generate a custom numeric range (overrides `count` when `step` is set).
`precision`	int (default `0`)	Decimal places for generated numeric values.
`orientation`	`horizontal` (default) / `vertical`	Layout direction.

`magnification`	float (default `2`)	Maximum magnification factor under the cursor.
`lensRange`	int (default `3`)	How many adjacent items the lens influences.
`itemSize`	string px (default `24`)	Base size of each item.
`gap`	string px (default `10`)	Gap between items.
`selectionMode`	`click` (default) / `hover`	How an item is selected.
`easing`	`linear` / `ease-out` / `ease-in-out` / `spring` / `cubic-bezier(…)`	Transition easing.
`transitionDuration`	int ms (default `200`)	Transition duration.
`pillColor` / `hoverColor` / `activeColor`	Mantine color	Default-pill colors for the inactive / hovered / active states.
`pillRadius`	Mantine radius (default `xl`)	Default-pill corner radius.
`ariaLabel`	string	Accessible label for the component.
`withScale`	`true` (default) / `false`	Scale items on hover.
`withOpacity`	`true` / `false` (default `false`)	Fade distant items.
`withBlur`	`true` / `false` (default `false`)	Blur distant items.
`expandOnHover`	`true` / `false` (default `false`)	Push neighbours apart on magnify (Dock style).
`withWheel`	`true` / `false` (default `false`)	Enable mouse-wheel navigation.
`loop`	`true` / `false` (default `false`)	Wrap-around navigation.
`withIndicator`	`true` / `false` (default `false`)	Show a marker beside the active item.

`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

CSS Selector

The island mounts under a stable wrapper you can target from your own CSS:

```
[data-aardvark-island="LensSelect"] {
  /* your overrides */
}
```

The upstream component also exposes Mantine Styles-API class names (`root`, `track`, `item`, `itemContent`, `itemPill`, `indicator`) and CSS variables (`--ls-item-size`, `--ls-magnification`, `--ls-range`, `--ls-gap`, `--ls-pill-color`, `--ls-easing`, …) for finer control.

Injecting Attributes

Use `attr={…}` to forward raw HTML attributes onto the rendered root element — handy for analytics hooks, test ids, or ARIA attributes the component doesn't expose as a prop. The object passes through verbatim (it is not a React prop):

Source: Markdown

```
{% lensselect attr={'data-testid': 'dock', 'data-analytics': 'launcher'} count=4 %}
```

Source: Python

```
component('aardvark', 'lensselect', attr={'data-testid': 'dock', 'data-analytics':
'launcher'}, count=4)
```

ListViewTable

`listviewtable` renders a **Finder-style list-view table** — rows of records under sortable column headers, with optional **drag-to-reorder** and **drag-to-resize** columns. It hydrates into a fully interactive island in the browser, with no JavaScript to write.

You pass two JSON **strings**: `data`, an array of row record objects, and `columns`, an array of `{key, title, sortable, width, textAlign, sticky}` column definitions. The tag parses both at build time and hands them to the table. Use it as `{% listviewtable %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'listviewtable', …)`.

A **Community Component** — wraps ListViewTable by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-list-view-table`.

Demonstrations

Basic list view

A `data` array of records and a `columns` array of definitions. Each column's `key` selects the field to display; `title` labels the header and `sortable` lets the reader sort by it:

```
{% listviewtable
data='[{"name":"Documents","kind":"Folder","size":"--"},{"name":"README.md","kind":"Markdown
","size":"2.1 KB"},{"name":"package.json","kind":"JSON","size":"1.8
KB"},{"name":"src","kind":"Folder","size":"--"}]'
columns='[{"key":"name","title":"Name","sortable":true},{"key":"kind","title":"Kind","sortab
le":true,"width":140},{"key":"size","title":"Size","sortable":true,"width":120,"textAlign":"
right"}]' %}
```

Borders, striping, and hover highlight

Turn on `withTableBorder`, `withColumnBorders`, `striped`, and `highlightOnHover` for a more table-like, scannable look:

```
{% listviewtable data='[{"name":"Ada
Lovelace","role":"Admin","status":"Active"},{"name":"Alan
Turing","role":"Editor","status":"Active"},{"name":"Grace
Hopper","role":"Viewer","status":"Invited"}]'
columns='[{"key":"name","title":"Name","sortable":true},{"key":"role","title":"Role","sortab
le":true,"width":140},{"key":"status","title":"Status","width":140}]' withTableBorder=true
withColumnBorders=true striped=true highlightOnHover=true %}
```

Reorderable and resizable columns

`enableColumnReordering` lets readers drag column headers to reorder them; `enableColumnResizing` lets them drag the column edges to resize. Set a `height` to cap the table and scroll the body:

```
{% listviewtable data='[{"id":1,"task":"Write
docs","owner":"Ada","due":"2024-06-01"},{"id":2,"task":"Ship
release","owner":"Alan","due":"2024-06-04"},{"id":3,"task":"Review
PRs","owner":"Grace","due":"2024-06-02"}]'
columns='[{"key":"task","title":"Task","sortable":true},{"key":"owner","title":"Owner","sort
able":true,"width":140},{"key":"due","title":"Due","sortable":true,"width":140,"textAlign":"
right"}]' enableColumnReordering=true enableColumnResizing=true withTableBorder=true %}
```

With other components

A list view sits naturally inside a card alongside other content — here a short text lead-in above a files table:

Recent files

```
{% card %}
{% text size='sm' c='dimmed' %}Recent files{% endText %}

{% listviewtable data='[{"name":"report.pdf","size":"482 KB"},{"name":"notes.txt","size":"3
KB"}]'
columns='[{"key":"name","title":"Name","sortable":true},{"key":"size","title":"Size","width"
:120,"textAlign":"right"}]' withTableBorder=true highlightOnHover=true %}
{% endCard %}
```

Attributes

Every attribute is optional except that a useful table needs both `data` and `columns`; omit any other to take its upstream default. The body is ignored — `listviewtable` is a viewer, not a container.

Attribute	Valid values	Description
`data`	JSON array string	The rows — an array of record objects. Parsed at build time; a malformed value warns and degrades to an empty array.

`columns`	JSON array string	The columns — an array of `{key, title, sortable, width, minWidth, maxWidth, textAlign, sticky, noWrap, ellipsis, hidden}` objects. `key` selects the record field; `title` labels the header.
`height`	CSS length (string)	Cap the table height; the body scrolls past it.
`rowKey`	String	Record field to use as the unique row key.
`borderRadius`	`xs`, `sm`, `md`, `lg`, `xl`	Outer container radius when `withTableBorder` is on (default `sm`).
`striped`	`true` / `false` (default `false`)	Stripe alternate rows.
`withTableBorder`	`true` / `false` (default `false`)	Draw a border around the table.
`withColumnBorders`	`true` / `false` (default `false`)	Draw borders between columns.
`withRowBorders`	`true` / `false` (default `true`)	Draw borders between rows.
`highlightOnHover`	`true` / `false` (default `false`)	Highlight a row on hover.
`stickyHeader`	`true` / `false` (default `false`)	Keep the header row sticky while the body scrolls.

`tabularNums`	`true` / `false` (default `false`)	Use tabular (fixed-width) numerals.
`enableColumnReordering`	`true` / `false` (default `false`)	Let readers drag column headers to reorder.
`enableColumnResizing`	`true` / `false` (default `false`)	Let readers drag column edges to resize.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element.

Per-column custom cell renderers (`renderCell` / `renderHeader`) are React functions, so they can't ride this JSON channel; columns here are plain field projections that display the record's value for each column `key`.

CSS Selector

The table is an island wrapper carrying `data-aardvark-island="ListViewTable"`; the upstream component owns the inner table chrome. Style or target it from your own CSS through the island attribute or any `attr`-supplied id/class:

```
/* The ListViewTable island wrapper */
[data-aardvark-island="ListViewTable"] {
  margin-block: var(--mantine-spacing-md);
}
```

The upstream package also exposes CSS variables (e.g. `--lvt-border-color`, `--lvt-shadow-color`, `--list-view-cell-font-size`) and Styles API selectors for the header, rows, cells, and sticky columns — see the upstream documentation for the full list.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (an `id`, `class`, `data-*`, ARIA attributes, inline event handlers) straight onto the rendered element — exactly like `component('aardvark', 'listviewtable', attr={…})`. These ride the `data-aardvark-attr` channel, not React props, so they don't collide with the component's own props:

Source: Markdown

```
{% listviewtable data='[{"name":"file.txt","size":"1 KB"}]'
columns='[{"key":"name","title":"Name"},{"key":"size","title":"Size"}]' attr={'id':
'demo-table', 'data-role': 'file-list'} %}
```

Source: Python

```
component('aardvark', 'listviewtable',
          data='[{"name":"file.txt","size":"1 KB"}]',
          columns='[{"key":"name","title":"Name"},{"key":"size","title":"Size"}]',
          attr={'id': 'demo-table', 'data-role': 'file-list'})
```

Audio

`{% audio %}` is a Mantine-native **audio player** — a play/pause transport, a scrubbable timeline, and volume and speed controls, built on the Web Audio API. Point it at a file with `src` and pick a `variant` and `size`.

A **Community Component** — wraps Audio by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-audio`.

The live player needs the Web Audio API, which only exists in a browser, so under server-side rendering, with JavaScript off, or for a screen reader it degrades to a plain, fully-functional native `<audio controls>` element using the same source — nothing is lost.

Use it as `{% audio %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'audio', …)`.

Demonstrations

Basic player

Set `src` to the audio file URL. Everything else takes the component's defaults.

Source: Markdown

```
{% audio src='/media/sample.mp3' %}
```

Source: Python

```
component('aardvark', 'audio', src='/media/sample.mp3')
```

Variants and size

`variant` (`overlay`, `minimal`, `floating`, `bordered`) changes the layout; `size` (`xs`, `sm`, `md`, `lg`, `xl`) scales the whole player.

Source: Markdown

```
{% audio src='/media/sample.mp3' variant='bordered' size='lg' %}
```

Source: Python

```
component('aardvark', 'audio', src='/media/sample.mp3', variant='bordered', size='lg')
```

Fallback source

`fallbackSrc` is used when the primary `src` can't be played (e.g. an unsupported codec).

Source: Markdown

```
{% audio src='/media/sample.mp3' fallbackSrc='/media/sample.ogg' %}
```

With other components

Drop a player into a Card to frame a downloadable sample alongside its description.

Episode 12 — Static Sites

A walkthrough of the build pipeline.

Source: Markdown

```
{% card title='Episode 12 — Static Sites' withBorder=true %}
A walkthrough of the build pipeline.

{% audio src='/media/sample.mp3' variant='minimal' %}
{% endCard %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `asBackground`) become `=True`.

Attribute	Valid values	Description
`src`	An audio file URL (string)	The audio source to play.
`variant`	`overlay` / `minimal` / `floating` / `bordered`	The player layout style.
`size`	`xs` / `sm` / `md` / `lg` / `xl`	Scales the whole player.
`radius`	A Mantine radius token or any CSS length	Corner rounding.
`asBackground`	`true` / `false` (default `false`)	Render as an ambient background track.
`shortcuts`	`true` / `false` (default `true`)	Enable keyboard transport controls.
`fallbackSrc`	An audio file URL (string)	Played if the primary `src` fails.

`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element (see below).

CSS Selector

The island wrapper carries a stable hook you can target from your own CSS:

```
[data-aardvark-audio] {
  /* your overrides */
}
```

The player's inner chrome is styled by the upstream `@gfazioli/mantine-audio` stylesheet (bundled automatically via a CSS `@import`), so its own Mantine-style class names apply too.

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered element — anything Mantine doesn't model as a prop (ARIA, `data-*`, analytics hooks). It rides a separate channel from the component props, so it never collides with them.

A common use is tagging the player with a `data-onboarding-tour-id` so the Onboarding tour can spotlight it:

Source: Markdown

```
{% audio src='/media/sample.mp3' attr={'data-onboarding-tour-id': 'player', 'aria-label':
'Episode 12 audio'} %}
```

Source: Python

```
component('aardvark', 'audio', src='/media/sample.mp3',
          attr={'data-onboarding-tour-id': 'player', 'aria-label': 'Episode 12 audio'})
```

Video

`{% video %}` is a Mantine-native **video player** built on the native `<video>` element — a play/pause transport, a scrubbable **timeline**, a mute toggle and volume slider, captions, **Picture-in-Picture**, keyboard shortcuts, and a fullscreen control. Point it at a file with `src`, set a `poster` thumbnail, and frame it with `aspectRatio`.

A **Community Component** — wraps Video by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-video`.

The live player wires up browser-only media APIs (Picture-in-Picture, fullscreen, MediaSession), so under server-side rendering, with JavaScript off, or for a screen reader it degrades to a plain, fully-functional native `<video controls>` element using the same source and poster — nothing is lost.

Use it as `{% video %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'video', …)`.

Demonstrations

Basic player

Set `src` to the video URL and `poster` to a thumbnail shown before playback.

Source: Markdown

```
{% video src='/media/sample.mp4' poster='/landscape.jpg' %}
```

Source: Python

```
component('aardvark', 'video', src='/media/sample.mp4', poster='/landscape.jpg')
```

Aspect ratio

`aspectRatio` sizes the frame — pass a number such as `1.777` for 16:9 or `1.333` for 4:3. The fallback box keeps the same ratio so the layout never jumps when the live player loads.

Source: Markdown

```
{% video src='/media/sample.mp4' poster='/landscape.jpg' aspectRatio=1.777 %}
```

Source: Python

```
component('aardvark', 'video', src='/media/sample.mp4',
          poster='/landscape.jpg', aspectRatio=16 / 9)
```

Variants and auto-hiding controls

`variant` (`overlay`, `minimal`, `floating`, `bordered`) sets the layout; `autoHideControls` takes a number of milliseconds — the control bar fades after that much inactivity during playback so it stays out of the way (use `0` to disable auto-hide).

Source: Markdown

```
{% video src='/media/sample.mp4' poster='/landscape.jpg' variant='bordered'
autoHideControls=3000 %}
```

Source: Python

```
component('aardvark', 'video', src='/media/sample.mp4',
          poster='/landscape.jpg', variant='bordered', autoHideControls=3000)
```

With other components

Frame a clip in a Card with a heading and description.

Product tour

A two-minute walkthrough of the editor.

Source: Markdown

```
{% card title='Product tour' withBorder=true %}
A two-minute walkthrough of the editor.

{% video src='/media/sample.mp4' poster='/landscape.jpg' aspectRatio=1.777 variant='minimal'
%}
{% endCard %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `asBackground`) become `=True`.

Attribute	Valid values	Description
`src`	A video file URL (string)	The video source to play.
`poster`	An image URL (string)	Thumbnail shown before playback.

`aspectRatio`	A positive number (e.g. `1.777`)	Frame aspect ratio (width ÷ height).
`variant`	`overlay` / `minimal` / `floating` / `bordered`	The player layout style.
`radius`	A Mantine radius token or any CSS length	Corner rounding.
`controls`	`true` / `false` (default `true`)	Show or hide the control bar.
`shortcuts`	`true` / `false` (default `true`)	Enable keyboard controls (Space, arrows, M, F, P).
`autoHideControls`	A number of milliseconds (e.g. `3000`; `0` disables)	Inactivity delay before the control bar fades during playback.
`asBackground`	`true` / `false` (default `false`)	Render as a section-background video.
`fallbackSrc`	A video file URL (string)	Played if the primary `src` fails.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element (see below).

CSS Selector

The island wrapper carries a stable hook you can target from your own CSS:

```
[data-aardvark-video] {
  /* your overrides */
}
```

The player's inner chrome is styled by the upstream `@gfazioli/mantine-video` stylesheet (bundled automatically via a CSS `@import`), so its own Mantine-style class names apply too.

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered element — anything Mantine doesn't model as a prop (ARIA, `data-*`, analytics hooks). It rides a separate channel from the component props, so it never collides with them.

A common use is tagging the player with a `data-onboarding-tour-id` so the Onboarding tour can spotlight it:

Source: Markdown

```
{% video src='/media/sample.mp4' poster='/landscape.jpg' attr={'data-onboarding-tour-id':
'player', 'aria-label': 'Product tour video'} %}
```

Source: Python

```
component('aardvark', 'video', src='/media/sample.mp4', poster='/landscape.jpg',
          attr={'data-onboarding-tour-id': 'player', 'aria-label': 'Product tour video'})
```

Onboarding

`{% onboarding %}` is a guided **product tour** — it dims the page, spotlights one element at a time behind a cutout, and walks the reader through your UI with a popover (a title and some content) per step. The body of the tag is the content the tour runs over; each target carries a `data-onboarding-tour-id` matching a step's `id`.

A **Community Component** — wraps Onboarding by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-onboarding-tour`.

A tour needs a "start" trigger and controlled state, which a static page can't wire up on its own, so the tag renders a **Start tour** button (relabel it with `triggerLabel`) that begins the tour and resets when it ends. The overlay, focus management, and element measurement are browser-only, so under server-side rendering or with JavaScript off the body content stays fully usable and the trigger is simply inert — nothing is hidden behind a tour that can't run.

Use it as `{% onboarding %} … {% endOnboarding %}` in Markdown, or — for a tour built from Python data — call `component('aardvark', 'onboarding', tour=[…], children=…)`.

Demonstrations

A two-step tour

Define the steps with `tour`, a JSON array of `{id, title, content}` objects. Mark each target in the body with a matching `data-onboarding-tour-id` using `attr={…}`. Click **Start tour** to run it.

Save

Share

Source: Markdown

```
{% onboarding tour='[{"id": "save", "title": "Save your work", "content": "This button saves
the current document."}, {"id": "share", "title": "Share it", "content": "Invite teammates
to collaborate."}]' %}
{% group %}
{% button attr={'data-onboarding-tour-id': 'save'} %}Save{% endButton %}
{% button variant='light' attr={'data-onboarding-tour-id': 'share'} %}Share{% endButton %}
{% endGroup %}
{% endOnboarding %}
```

Source: Python

```
component('aardvark', 'onboarding',
          tour=[
              {'id': 'save', 'title': 'Save your work',
               'content': 'This button saves the current document.'},
              {'id': 'share', 'title': 'Share it',
```

```
                'content': 'Invite teammates to collaborate.'},
            ],
            children='…the UI the tour runs over…')
```

Custom trigger label and hiding the stepper

`triggerLabel` sets the button text. The popover shows a step indicator by default; pass `withStepper=false` to hide it.

Edit profile

Source: Markdown

```
{% onboarding triggerLabel='Take the tour' withStepper=false tour='[{"id": "edit", "title":
"Edit inline", "content": "Click any field to edit it in place."}]' %}
{% button attr={'data-onboarding-tour-id': 'edit'} %}Edit profile{% endButton %}
{% endOnboarding %}
```

With other components

The tour runs over whatever you put in its body, so it composes with any built-in. Here it spotlights cards in a CardGrid.

Metrics

Usage and trends.

Activity

The latest events.

Source: Markdown

```
{% onboarding triggerLabel='Tour the dashboard' tour='[{"id": "metrics", "title": "Your
metrics", "content": "Live numbers update here."}, {"id": "activity", "title": "Recent
activity", "content": "See what changed lately."}]' %}
{% cardGrid cols=2 %}
{% card title='Metrics' withBorder=true attr={'data-onboarding-tour-id': 'metrics'} %}Usage
and trends.{% endCard %}
{% card title='Activity' withBorder=true attr={'data-onboarding-tour-id': 'activity'} %}The
latest events.{% endCard %}
{% endCardGrid %}
{% endOnboarding %}
```

Attributes

Omit any attribute to take its default. The popover controls are all on by default; turn one off with `=false` (e.g. `withStepper=false`).

Attribute	Valid values	Description
`tour`	A JSON array of `{id, title, content}` objects	The tour steps; each `id` matches a target's `data-onboarding-tour-id`. A malformed value warns at build time and the tour renders with no steps.
`triggerLabel`	A string (default `Start tour`)	The opener button's text.
`withSkipButton`	`true` / `false` (default `true`)	Show a "skip" control in the popover.
`withPrevButton`	`true` / `false` (default `true`)	Show the "previous step" control.
`withNextButton`	`true` / `false` (default `true`)	Show the "next step" control.
`withStepper`	`true` / `false` (default `true`)	Show a step indicator in the popover. Set `=false` to hide it.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered wrapper (see below).

For tours built from data — or to pass the full upstream API (lifecycle callbacks, per-step `cutoutPadding` / `cutoutRadius`, …) — call `component('aardvark', 'onboarding', tour=[…], …)` from Python; every extra keyword is forwarded straight to the underlying component.

CSS Selector

The island wrapper carries a stable hook you can target from your own CSS:

```
[data-aardvark-onboarding] {
  /* your overrides */
}
```

The overlay, the focus cutout, and the popover are portaled to `document.body` and styled by the upstream `@gfazioli/mantine-onboarding-tour` stylesheet (bundled automatically via a CSS `@import`), so its own Mantine-style class names apply too.

Injecting Attributes

`attr={…}` forwards raw HTML attributes straight onto the rendered wrapper — anything Mantine doesn't model as a prop (ARIA, `data-*`, analytics hooks). It rides a separate channel from the component props, so it never collides with them.

The same `attr={…}` channel is how you mark a **tour target**: put `data-onboarding-tour-id` on any element inside the body so a step's `id` can find it.

Get started

Source: Markdown

```
{% onboarding tour='[{"id": "cta", "title": "Get started", "content": "Click here to
begin."}]' attr={'data-section': 'hero-tour'} %}
{% button attr={'data-onboarding-tour-id': 'cta'} %}Get started{% endButton %}
{% endOnboarding %}
```

Source: Python

```
component('aardvark', 'onboarding',
          tour='[{"id": "cta", "title": "Get started", "content": "Click here to begin."}]',
          attr={'data-section': 'hero-tour'},
          children=component('aardvark', 'button', children='Get started',
                             attr={'data-onboarding-tour-id': 'cta'}))
```

DataTable

A feature-rich **data grid** built on top of `@mantine/core`. Give it `columns` (a JSON array of column definitions) and `records` (a JSON array of row objects), and it renders a clean, themed, semantic table — server-side for the static rows, hydrating into an interactive island for sorting and selection.

A **Community Component** — wraps Mantine DataTable by **icflorescu**, **MIT** licensed, npm `mantine-datatable`.

Use it as `{% datatable … %}` in Markdown (it's self-closing — it takes no body), or call it from Python logic (loops, snippets) via `component('aardvark', 'datatable', …)`.

Demonstrations

Each `columns` entry has an `accessor` naming a field on the row objects in `records`; give it a `title` to override the auto-generated header, and `textAlign` / `width` to shape the column.

Source: Markdown

```
{% datatable withTableBorder=true columns='[
  {"accessor":"name","title":"Name"},
  {"accessor":"role","title":"Role"},
  {"accessor":"commits","title":"Commits","textAlign":"right"}
]' records='[
  {"name":"Ada Lovelace","role":"Maintainer","commits":482},
  {"name":"Alan Turing","role":"Core","commits":351},
  {"name":"Grace Hopper","role":"Core","commits":297}
]' %}
```

Source: Python

```
component('aardvark', 'datatable', withTableBorder=True,
          columns='[{"accessor":"name","title":"Name"}, …]',
          records='[{"name":"Ada Lovelace","role":"Maintainer","commits":482}, …]')
```

Striped, with hover highlight

Turn on `striped` for zebra rows and `highlightOnHover` so the row under the cursor stands out. Mark a column `sortable` and the header becomes clickable once the grid hydrates in the browser.

Source: Markdown

```
{% datatable withTableBorder=true striped=true highlightOnHover=true columns='[
  {"accessor":"product","title":"Product","sortable":true},
  {"accessor":"region","title":"Region"},
  {"accessor":"units","title":"Units","textAlign":"right","sortable":true}
```

```
]' records='[
  {"product":"Aardvark Pro","region":"NA","units":1280},
  {"product":"Aardvark Pro","region":"EU","units":960},
  {"product":"Aardvark Team","region":"NA","units":540},
  {"product":"Aardvark Team","region":"APAC","units":410}
]' %}
```

With other components

A DataTable sits naturally inside a Card or Paper surface — the table inherits the theme, and the surface gives it a titled frame.

Top contributors

Source: Markdown

```
{% card title="Top contributors" withBorder=true %}
{% datatable columns='[
  {"accessor":"name","title":"Name"},
  {"accessor":"commits","title":"Commits","textAlign":"right"}
]' records='[
  {"name":"Ada Lovelace","commits":482},
  {"name":"Alan Turing","commits":351}
]' %}
{% endCard %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `striped`) become `=True`.

Attribute	Valid values	Description
`columns`	JSON array of column defs	Each `{accessor, title?, textAlign?, sortable?, width?}`; `accessor` names the field on a record row. Required for a non-empty table.
`records`	JSON array of row objects	The rows to render, one per table row, keyed by each column's `accessor`.
`withTableBorder`	`true` / `false` (default `false`)	Draw a border around the whole table.

`withColumnBorders`	`true` / `false` (default `false`)	Draw vertical borders between columns.
`striped`	`true` / `false` (default `false`)	Zebra-stripe the rows.
`highlightOnHover`	`true` / `false` (default `false`)	Highlight the row under the cursor.
`noHeader`	`true` / `false` (default `false`)	Hide the column header row.
`height`	string (e.g. `400`, `60vh`)	Fix the table height; the body scrolls past it.
`borderRadius`	`xs` / `sm` / `md` / `lg` / `xl` or a size	Round the table corners.
`noRecordsText`	string	Text shown when `records` is empty.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element (see below).

CSS Selector

The tag mounts an island wrapper you can target directly. The grid itself carries `mantine-datatable`'s own classes, but everything sits under the island marker:

```
/* The DataTable island wrapper */
[data-aardvark-island="DataTable"] {
  /* … */
}
```

`mantine-datatable` ships its own stylesheet (pulled in automatically via a CSS `@import` in the component's co-located stylesheet), and the grid reads Mantine's theme CSS variables, so it matches the rest of the site out of the box.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes (data-, aria-, role, …) onto the rendered element, exactly like `component('DataTable', attr={…})`. These ride a separate channel from the React props above, so they reach the DOM untouched.

Source: Markdown

```
{% datatable attr={'data-role': 'contributors-grid', 'aria-label': 'Contributors'}
columns='[
  {"accessor":"name","title":"Name"}
]' records='[{"name":"Ada Lovelace"}]' %}
```

Source: Python

```
component('aardvark', 'datatable',
          attr={'data-role': 'contributors-grid', 'aria-label': 'Contributors'},
          columns='[{"accessor":"name","title":"Name"}]',
          records='[{"name":"Ada Lovelace"}]')
```

BlockNote

BlockNote is a **block-based rich-text editor** by **TypeCellOS** — a Notion-style WYSIWYG editor built on ProseMirror, with a first-class Mantine theme (`@blocknote/mantine`). It's a natural fit for a "Community Component" wrapper on paper.

Aardvark, however, does **not** ship BlockNote as a built-in `{% tag %}`. This page exists to record why, and to point you at the project so you can integrate it yourself if its license works for your project.

Not bundled — license verdict

BlockNote's source code is licensed under the **Mozilla Public License 2.0 (MPL-2.0)**, a **copyleft** license. Aardvark's policy for Community Components is to bundle only permissively-licensed projects (MIT / Apache-2.0 / BSD / ISC) into its island bundle. Because MPL-2.0 carries file-level copyleft obligations, BlockNote is **documented but not bundled**. There is no `{% blocknote %}` tag.

License

The verdict was confirmed against two independent sources on 2026-06-19:

Source	Finding
`registry.npmjs.org/@blocknote/core`	`"license": "MPL-2.0"` (and likewise for `@blocknote/mantine`, `@blocknote/react`)
`LICENSE.txt` in the GitHub repo	"Source code in this repository is covered by the Mozilla Public License Version 2.0 (MPL-2.0), except for the XL packages."

A few details that matter for the verdict:

- **The main packages are MPL-2.0.** `@blocknote/core`, `@blocknote/mantine`, and `@blocknote/react` — the packages a wrapper would use — are all **MPL-2.0**.
- **The `@blocknote/xl-*` packages are GPL-3.0** (with a separate commercial license offered). These are an even stronger copyleft and are likewise excluded.
- **MPL-2.0 is not anti-commercial.** It is OSI-approved and permits commercial use and redistribution. Its copyleft is file-level: only changes to the MPL-licensed source files themselves must be shared back. That makes it materially different from AGPL or "source-available" licenses.

Aardvark's Community Component bundling bar is intentionally narrow — **permissive only** — so that the island bundle (and anything built on top of it) stays free of file-level copyleft obligations. MPL-2.0, while friendly, sits on the copyleft side of that line, so BlockNote is not bundled. This is a packaging decision, **not** a statement that BlockNote is unsuitable — it's an excellent editor.

Using BlockNote in your own project

Nothing stops you from adopting BlockNote directly — MPL-2.0 is perfectly usable in a commercial product. Install it into your own front-end build and follow the upstream Getting Started guide:

```
npm install @blocknote/core @blocknote/mantine @blocknote/react
```

The Mantine theme ships its own stylesheet, imported via the package's `style.css` export:

```
import "@blocknote/core/fonts/inter.css";
import "@blocknote/mantine/style.css";
```

A minimal Mantine-themed editor looks like this upstream:

```
import { useCreateBlockNote } from "@blocknote/react";
import { BlockNoteView } from "@blocknote/mantine";

function Editor() {
  const editor = useCreateBlockNote();
  return <BlockNoteView editor={editor} />;
}
```

For a **read-only** view (the shape that would suit a static docs page), upstream exposes an `editable={false}` prop on `BlockNoteView`. See the BlockNote docs for the full API.

Links

- Project site: https://www.blocknotejs.org/
- Documentation: https://www.blocknotejs.org/docs
- Source: https://github.com/TypeCellOS/BlockNote
- npm: `@blocknote/core` · `@blocknote/mantine` · `@blocknote/react`
- License: MPL-2.0

A **Community Component** entry — BlockNote by **TypeCellOS**, **MPL-2.0** licensed, npm **`@blocknote/*`**. Documented but **not bundled** by Aardvark because MPL-2.0 is a copyleft license; Aardvark bundles only permissively-licensed (MIT / Apache-2.0 / BSD / ISC) components.

Community Marquee

`{% communityMarquee %}` is a scrolling ticker with extra flair — content that slides continuously across the row (logos, announcements, a tagline). On top of the left/right/up/down scroll of the built-in `{% marquee %}`, this variant adds isometric and circle layouts, fade-edge modes, repeat control, and 3D tilt/perspective/rotate/skew.

> **A Community Component** — wraps Marquee by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-marquee`.

Because Mantine core already ships a native `Marquee` (the one behind the built-in `{% marquee %}`), the community variant is registered under the distinct tag **`communityMarquee`** so the two never collide — both are available.

Use it as `{% communityMarquee %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'communityMarquee', …)`.

Demonstrations

Basic ticker

The block body is the scrolling content. `direction` (left, right, up, down) sets the scroll axis; `duration` is the seconds for one loop.

Ship docs faster · Built-in components · No JavaScript to write · Markdown all the way down ·

Source: Markdown

```
{% communityMarquee duration=20 %}
Ship docs faster · Built-in components · No JavaScript to write · Markdown all the way down
·
{% endCommunityMarquee %}
```

Source: Python

```
component('aardvark', 'communityMarquee', duration=20,
          children='Ship docs faster · Built-in components · No JavaScript to write ·')
```

Pause on hover and fade edges

`pauseOnHover` stops the scroll while the pointer is over it; `fadeEdges` softens the edges so content fades in and out instead of clipping (set it to a bare flag for the default fade, or name a mode — `linear`, `ellipse`, `rect`).

React · Mantine · Aardvark · Static sites · Islands · Markdown ·

Source: Markdown

```
{% communityMarquee duration=25 pauseOnHover=true fadeEdges='linear' %}
React · Mantine · Aardvark · Static sites · Islands · Markdown ·
{% endCommunityMarquee %}
```

Reverse direction

`direction='right'` scrolls the other way; `direction='up'` / `direction='down'` scroll vertically.

← scrolling the other way · keep it moving · ← scrolling the other way ·

Source: Markdown

```
{% communityMarquee direction='right' duration=20 %}
← scrolling the other way · keep it moving ·
{% endCommunityMarquee %}
```

Isometric variant

`variant='isometric'` tilts the row into a 3D plane; tune it with `tilt`, `perspective`, `rotate`, and `skew`.

Isometric · 3D ticker · Tilt and perspective · Isometric · 3D ticker ·

Source: Markdown

```
{% communityMarquee variant='isometric' duration=22 tilt=45 %}
Isometric · 3D ticker · Tilt and perspective ·
{% endCommunityMarquee %}
```

With other components

The scrolling content can be any Aardvark tags, already rendered — here a row of Badges glides across as a feature ticker.

[Fast] [Themeable] [Accessible] [Open]

Source: Markdown

```
{% communityMarquee duration=18 gap='1.5rem' %}
{% badge variant='light' color='grape' %}Fast{% endBadge %} {% badge variant='light'
color='indigo' %}Themeable{% endBadge %} {% badge variant='light' color='teal'
%}Accessible{% endBadge %}
{% endCommunityMarquee %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `pauseOnHover`, `fadeEdges`) become `=True`.

Attribute	Valid values	Description
`direction`	`left` / `right` / `up` / `down` (default `left`)	Scroll axis and direction. Maps to the package's `vertical` (up/down) + `reverse` (right/down).
`variant`	`default` / `isometric` / `circle`	Layout variant. `isometric` and `circle` add a 3D / ring presentation.
`duration`	Integer (seconds)	Seconds for one full loop.
`gap`	A CSS length (e.g. `2rem`)	Space between the looping copies of the content.
`repeat`	Integer	How many times the content is cloned to fill the row.
`pauseOnHover`	`true` / `false` (default `false`)	Pause scrolling while the pointer is over the marquee.
`fadeEdges`	bare flag / `linear` / `ellipse` / `rect`	Fade the leading/trailing edges. A bare flag uses the default fade; a named value picks the mask shape.
`fadeEdgesSize`	`xs`–`xl` or a percentage	Size of the edge fade.
`tilt`	Integer (degrees)	3D tilt angle (isometric variant).
`perspective`	Integer (px)	Depth of the 3D scene (3D variants).
`rotate`	Integer (degrees)	In-plane rotation (isometric variant).
`skew`	Integer (degrees)	Horizontal skew (isometric variant).
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element (see Injecting Attributes).

CSS Selector

The mounted island is wrapped in an element carrying `data-aardvark-island="CommunityMarquee"`, so you can target the whole ticker from your own CSS:

```
[data-aardvark-island='CommunityMarquee'] {
```

```
  margin-block: 2rem;
}
```

The package itself styles the track and fade with its own data-attribute selectors (`[data-variant]`, `[data-vertical]`, `[data-fade-edges]`) and CSS variables (`--marquee-gap`, `--marquee-fade-edge-size`, …); the stylesheet ships from `@gfazioli/mantine-marquee/styles.css`, imported automatically by the island.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes onto the rendered element — useful for `id`, `data-*` hooks, ARIA, or analytics attributes that aren't component props:

Tagged for analytics · Accessible label · Tagged for analytics ·

Source: Markdown

```
{% communityMarquee duration=20 attr={'data-analytics': 'home-ticker', 'aria-label':
'Feature ticker'} %}
Tagged for analytics · Accessible label ·
{% endCommunityMarquee %}
```

Source: Python

```
component('aardvark', 'communityMarquee', duration=20,
          attr={'data-analytics': 'home-ticker', 'aria-label': 'Feature ticker'},
          children='Tagged for analytics · Accessible label ·')
```

Text Animate

`{% textAnimate %}` animates a run of text — revealing it (or looping it) with an effect, broken up by character, word, or line. Reach for it on a hero headline, a tagline, or anywhere a little motion earns its keep.

> **A Community Component** — wraps TextAnimate by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-text-animate`.

Use it as `{% textAnimate %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'textAnimate', …)`. The text is the block body or the plain-text `text` param.

Demonstrations

Basic reveal

`animate='in'` reveals the text once; `animation` picks the effect (fade, blur, scale, slideUp, slideDown, slideLeft, slideRight, …); `by` sets the granularity (text, word, character, line).

Documentation that moves with you

Source: Markdown

```
{% textAnimate animate='in' animation='fade' by='word' %}
Documentation that moves with you
{% endTextAnimate %}
```

Source: Python

```
component('aardvark', 'textAnimate',
          animate='in', animation='fade', by='word',
          children='Documentation that moves with you')
```

Looping animation

`animate='loop'` runs the effect continuously; `loopDelay` (ms) is the pause between cycles. Per-character blur makes a nice attention-getter.

Always shipping

Source: Markdown

```
{% textAnimate animate='loop' animation='blur' by='character' loopDelay=1200 %}
Always shipping
{% endTextAnimate %}
```

Trigger on scroll

`trigger='inView'` holds the animation until the text scrolls into the viewport, so the reveal plays as the reader reaches it. `duration` and `delay` (seconds) tune the timing.

This line slides up the moment it enters the viewport.

Source: Markdown

```
{% textAnimate animate='in' animation='slideUp' by='line' trigger='inView' duration=0.6 %}
This line slides up the moment it enters the viewport.
{% endTextAnimate %}
```

With other components

Drop an animated headline inside a Paper surface to build a hero card — the animation runs on the text while the surface stays put.

Build. Document. Ship.

Source: Markdown

```
{% paper withBorder=true p='xl' radius='md' %}
{% textAnimate animate='in' animation='scale' by='word' %}
Build. Document. Ship.
{% endTextAnimate %}
{% endPaper %}
```

Attributes

Omit any attribute to take its default.

Attribute	Valid values	Description
`animate`	`in` / `out` / `loop` / `static` / `none`	Animation mode. `in` reveals once, `loop` runs continuously, `static` shows the text without motion.
`animation`	`fade` / `blur` / `scale` / `slideUp` / `slideDown` / `slideLeft` / `slideRight` (plus `*Elastic`, `blurUp`, `blurDown`)	The effect applied to each segment.

`by`	`text` / `word` / `character` / `line`	Granularity the text is split into for the animation.
`duration`	Number (seconds)	Length of the animation.
`delay`	Number (seconds)	Delay before the animation starts.
`segmentDelay`	Number (seconds)	Stagger between consecutive segments.
`loopDelay`	Integer (ms)	Pause between loop cycles (`animate='loop'`).
`trigger`	`inView`	Start when the text scrolls into view, instead of immediately.
`text`	string	Plain-text alternative to the block body (HTML-escaped). The block body wins when both are set.
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element (see Injecting Attributes).

CSS Selector

The mounted island is wrapped in an element carrying `data-aardvark-island="TextAnimate"`, so you can target it from your own CSS:

```
[data-aardvark-island='TextAnimate'] {
  font-size: 1.5rem;
}
```

The package drives the animation through its own data-attribute selectors (`[data-text-animate]`, `[data-text-animate-animation]`, `[data-animating]`, …) and CSS variables; the stylesheet ships from `@gfazioli/mantine-text-animate/styles.css`, imported automatically by the island.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes onto the rendered element — useful for `id`, `data-*` hooks, ARIA, or analytics attributes that aren't component props:

Tagged headline

Source: Markdown

```
{% textAnimate animate='in' animation='fade' by='word' attr={'id': 'hero-headline',
'data-role': 'headline'} %}
Tagged headline
{% endTextAnimate %}
```

Source: Python

```
component('aardvark', 'textAnimate',
          animate='in', animation='fade', by='word',
          attr={'id': 'hero-headline', 'data-role': 'headline'},
          children='Tagged headline')
```

Border Animate

`{% borderAnimate %}` wraps any content in an **animated border** — a beam that travels around the edge, a diffused glow, or a subtle pulse. It's a nice way to draw the eye to a card, a call-to-action, or a highlighted note.

> **A Community Component** — wraps BorderAnimate by **gfazioli**, **MIT** licensed, npm `@gfazioli/mantine-border-animate`.

The effect respects `prefers-reduced-motion`, so readers who opt out of motion see a static border. Use it as `{% borderAnimate %}` in Markdown, or call it from Python logic (loops, snippets) via `component('aardvark', 'borderAnimate', …)`. The wrapped content is the block body.

Demonstrations

Beam

`variant='beam'` (the default) sends a glowing beam traveling around the border. `duration` (seconds) sets the lap time; `colorFrom` / `colorTo` set the gradient.

A beam traces this card's border.

Source: Markdown

```
{% borderAnimate variant='beam' duration=4 radius='md' colorFrom='indigo' colorTo='cyan' %}
{% paper p='lg' radius='md' %}A beam traces this card's border.{% endPaper %}
{% endBorderAnimate %}
```

Source: Python

```
component('aardvark', 'borderAnimate',
          variant='beam', duration=4, radius='md',
          colorFrom='indigo', colorTo='cyan',
          children=component('aardvark', 'paper', p='lg', radius='md',
                             children="A beam traces this card's border."))
```

Glow

`variant='glow'` surrounds the content with a soft, pulsating light. `blur` (xs–xl) controls how diffuse it is.

A glow pulses around this card.

Source: Markdown

```
{% borderAnimate variant='glow' duration=3 radius='md' blur='md' colorFrom='grape'
colorTo='pink' %}
```

```
{% paper p='lg' radius='md' %}A glow pulses around this card.{% endPaper %}
{% endBorderAnimate %}
```

Pulse, paused on hover

`variant='pulse'` scales the border in and out; `pauseOnHover` freezes it while the pointer is over the content.

Hover to pause the pulse.

Source: Markdown

```
{% borderAnimate variant='pulse' duration=2 radius='md' pauseOnHover=true colorFrom='teal'
colorTo='lime' %}
{% paper p='lg' radius='md' %}Hover to pause the pulse.{% endPaper %}
{% endBorderAnimate %}
```

With other components

Wrap any Aardvark content — here a Badge gets a beam border to mark a featured tier.

[Featured]

Source: Markdown

```
{% borderAnimate variant='beam' duration=3 radius='xl' colorFrom='orange' colorTo='red' %}
{% badge variant='filled' color='orange' size='xl' %}Featured{% endBadge %}
{% endBorderAnimate %}
```

Attributes

Omit any attribute to take its default. Bare flags (e.g. `pauseOnHover`) become `=True`; `withMask` and `animate` default on, so set them to `false` to turn them off.

Attribute	Valid values	Description
`variant`	`beam` / `glow` / `pulse` (default `beam`)	The animation style.
`beamMode`	`path` / `conic` (default `path`)	How the beam is rendered (beam variant).
`size`	`xs`–`xl`	Beam / effect size.

`duration`	Number (seconds)	Animation speed (lap time).
`colorFrom`	A color (token or CSS color)	Start of the gradient.
`colorTo`	A color (token or CSS color)	End of the gradient.
`radius`	A Mantine radius token or CSS length	Border radius of the effect.
`blur`	`xs`–`xl`	Softness of the glow (glow variant).
`angle`	Integer (degrees)	A static rotation angle for the border.
`borderWidth`	A CSS length (e.g. `2px`)	Thickness of the animated border.
`borderOpacity`	Number `0`–`1`	Opacity of the effect.
`withMask`	`true` / `false` (default `true`)	Clip the effect to the border (vs. fill behind the content).
`pauseOnHover`	`true` / `false` (default `false`)	Pause the animation while the pointer is over the content.
`animate`	`true` / `false` (default `true`)	Enable the animation (set `false` for a static border).
`attr={…}`	An object of HTML attributes	Forwards raw HTML attributes onto the rendered element (see Injecting Attributes).

CSS Selector

The mounted island is wrapped in an element carrying `data-aardvark-island="BorderAnimate"`, so you can target it from your own CSS:

```
[data-aardvark-island='BorderAnimate'] {
  display: inline-block;
}
```

The package draws the border through its own data-attribute selectors (`[data-variant]`, `[data-beam-mode]`, `[data-animate]`, `[data-with-mask]`, …) and CSS variables (`--border-animate-angle`, `--border-animate-blur`, …); the stylesheet ships from `@gfazioli/mantine-border-animate/styles.css`, imported automatically by the island.

Injecting Attributes

Pass `attr={…}` to forward raw HTML attributes onto the rendered element — useful for `id`, `data-*` hooks, ARIA, or analytics attributes that aren't component props:

Tagged for analytics.

Source: Markdown

```
{% borderAnimate variant='beam' duration=4 radius='md' attr={'data-role': 'cta-frame'} %}
{% paper p='lg' radius='md' %}Tagged for analytics.{% endPaper %}
{% endBorderAnimate %}
```

Source: Python

```
component('aardvark', 'borderAnimate',
          variant='beam', duration=4, radius='md',
          attr={'data-role': 'cta-frame'},
          children=component('aardvark', 'paper', p='lg', radius='md',
                             children='Tagged for analytics.'))
```

Aardvark Gateway API

The reference below is the **real, user-facing API of the Aardvark cloud gateway** — the metered OpenRouter proxy and dashboard backend behind the reader assistant. Chat, reader telemetry, dashboard analytics, account and API-key management, team, billing, and magic-link auth are all here, rendered inline by the `{% openapi %}` directive on an ordinary Markdown page set to `mode: wide` — no dedicated page generator, just a component.

It also doubles as the headline demonstration of that directive: every operation below is wired into the left-hand nav, with parameter and response tables, multi-language request samples, and a **Try it now** form. To splice a **single endpoint** into a page instead of the whole spec, see OpenAPI under Components.

The spec it renders, `openapi/aardvark-gateway.json`, isn't hand-written: it's **extracted from the gateway's own source** — every route's auth, request fields, query params, response shape, and error codes — and regenerated on every change, so this reference can't drift from the code it documents. (That's the same drift-proof loop you'd wire up for your own API.)

Aardvark Gateway API

v1.0.0

User-facing HTTP API of the Aardvark cloud gateway — the metered OpenRouter proxy and dashboard backend. Operator-only endpoints (`/admin/*`, `/operator-auth/*`, `/operator-api/*`) and the dashboard / operator SPA shells (`/dashboard*`, `/operator*`) are excluded by design.

This document is GENERATED from the gateway source: `scripts/gen-openapi.mjs` extracts every route's auth, request fields, query params, response shape, and error codes directly from the handlers, then renders them. Do NOT edit `openapi.json` (or hand-author its structure) — change the code, and regenerate. Only prose lives in `scripts/openapi/descriptions.mjs`.

Base URL: `https://gateway.aardvarkdocs.com`, `https://aardvark-gateway.{subdomain}.workers.dev`

`GET /`

Health check

Unauthenticated root health check; returns a fixed `text/plain` body.

Responses

Code	Description
200	aardvark-gateway ok

`POST /auth/logout`

Destroy the session

Deletes the session row and clears the cookie. Requires the session cookie and the `X-Aardvark-Dashboard` CSRF header. Idempotent.

Responses

Code	Description
`200`	Success.
`403`	Forbidden — `csrf`.
`500`	Server error — `logout_failed`.

`GET /auth/me`

Current user & teams

Returns the authenticated user, the active account, the role there, and all memberships. Returns 401 `{ authenticated: false }` when there is no valid session.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated.

`POST /auth/request-link`

Send a sign-in link

Emails a single-use magic link to the address. Neutral 200 regardless of whether the email exists (no account enumeration). Rate limited per IP and per email.

Request body

Field	Type	Required
`email`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`403`	Forbidden — `sso_required`.
`429`	Rate limited — `rate_limited`.

`POST /auth/saml/acs`

SAML ACS

The Assertion Consumer Service (HTTP-POST binding). Strictly validates the signed SAMLResponse against the account's pinned certificate (single signed assertion; enveloped-signature; exclusive-c14n digest; RSA-SHA256; audience/recipient/InResponseTo; replay cache), then mints the session. Only served when the gateway has SAML enabled — that disabled surface is the one failure here answered with the JSON error envelope (**404**); every other failure a browser can land on renders the styled SSO page instead.

Request body

Field	Type	Required
`SAMLResponse`	string	yes
`RelayState`	string	yes

Responses

Code	Description
`200`	An HTML page.
`400`	Bad request.
`403`	Forbidden.
`404`	Not found — `not_found`.
`413`	Payload too large.
`429`	Rate limited.

`GET /auth/saml/metadata`

SAML SP metadata

The SP EntityDescriptor XML the customer uploads to their IdP. Only served when the gateway has SAML enabled. The SP entity id is `<origin>/auth/saml/metadata?account=<id>`; the ACS is `<origin>/auth/saml/acs`.

Parameters

Name	In	Type	Required	Description
`account`	query	string	yes	The account id whose SP metadata to render.

Responses

Code	Description
`200`	An XML document (`application/xml`).
`400`	Bad request — `bad_request`.
`404`	Not found — `not_found`.
`429`	Rate limited — `rate_limited`.

`POST /auth/sso/begin`

Begin SSO login

Resolve the enabled IdP by the email's domain and return the next hop: for OIDC, `redirect_url` (an authorization URL with state + nonce + S256 PKCE); for SAML, `handoff_url` (a same-origin gateway page that auto-POSTs the AuthnRequest to the IdP under an IdP-scoped CSP). No session required — this is the pre-login surface.

Request body

Field	Type	Required
`email`	string	yes

Responses

Code	Description
`200`	Success.

`400`	Bad request — `bad_request`, `sso_misconfigured`.
`404`	Not found — `not_found`, `sso_not_configured`.
`429`	Rate limited — `rate_limited`.

`GET /auth/sso/callback`

OIDC callback

The OIDC redirect_uri. Consumes the single-use state, exchanges the code (client secret + PKCE), verifies the id_token (RS256 via JWKS; iss/aud/exp/iat/nonce; verified email), then mints the dashboard session and redirects to `/dashboard`. Renders an HTML result page.

Parameters

Name	In	Type	Required	Description
`code`	query	string		
`state`	query	string		

Responses

Code	Description
`200`	An HTML page.
`400`	Bad request.
`403`	Forbidden.
`429`	Rate limited.

`GET /auth/sso/saml-handoff`

SAML AuthnRequest handoff

The gateway-hosted SAML handoff page (SAML-enabled deployments only). The dashboard's `form-action 'none'` CSP can't POST the AuthnRequest to the IdP, so `begin` returns a `handoff_url` here; this page is a fresh document whose CSP permits form-action only to the configured IdP's SSO URL, and it auto-submits the AuthnRequest (`SAMLRequest` + `RelayState`). Reads the state without consuming it (the ACS consumes it on the IdP postback). Renders HTML.

Parameters

Name	In	Type	Required	Description
`rs`	query	string		

Responses

Code	Description
`200`	An HTML page.
`400`	Bad request.
`404`	Not found — `not_found`.

`POST /auth/switch`

Switch active account

Rotates the session to a different account the user belongs to. Requires the session cookie and the `X-Aardvark-Dashboard` CSRF header. Rate limited per user.

Request body

Field	Type	Required
`account_id`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `unauthenticated`.
`403`	Forbidden — `csrf`, `no_access`, `sso_required`, `sso_session_confined`.
`429`	Rate limited — `rate_limited`.

`GET /auth/verify`

Magic-link landing

Serves a tiny nonce-CSP HTML page that reads the token from the URL fragment and POSTs it to `/auth/verify`. The token never reaches the server as a query string.

Responses

Code	Description
`200`	An HTML page that completes the sign-in client-side.

`POST /auth/verify`

Redeem a magic link

The token in the body is the credential. On success sets the `__Host-av_session` cookie. NOTE: this endpoint's responses use the non-standard shape `{ ok, error }` (read by the landing page's script), not the usual error envelope.

Request body

Field	Type	Required
`token`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `invalid`.
`429`	Rate limited — `rate_limited`.

`GET /v1/activity`

Live activity rollup

Headline metrics over the last `days` (default 30, capped at 365): conversation/answer/feedback counts, verdict mix, uncertainty rate, engagement, support-deflection rate, and top languages.

Parameters

Name	In	Type	Required	Description
`days`	query	integer		Window length in days (default 30, capped at 365).

Responses

Code	Description
200	Success.
401	Unauthenticated — invalid_key.
403	Forbidden — forbidden_key.

`POST /v1/assistant/chat`

Support assistant turn

One metered turn of the dashboard support assistant. The gateway injects the system prompt (identity, plan-and-approve behavior, the docs index) and the account context, forces the latest Sonnet model, and forwards the conversation to the upstream model. Billed to the account (tagged as support spend), so it can return 402/429/503 like the chat endpoint. The reply may include `tool_calls` the dashboard renders for the user to approve before running.

Request body

Field	Type	Required
messages	array of item	yes

Responses

Code	Description
200	Success.
400	Bad request — bad_request.
401	Unauthenticated — invalid_key.
402	Payment required — anomaly_paused, grant_paused, insufficient_balance, no_billing_key.
403	Forbidden — csrf, forbidden_key.
413	Payload too large — too_large.
429	Rate limited — concurrency_limit, rate_limited.
502	Upstream error — upstream_error.

`503`	Service unavailable — `pricing_unavailable`, `unavailable`.

`GET /v1/assistant/doc`

Fetch a docs page

Server-side fetch of one Aardvark documentation page (Markdown) for the assistant's `fetch_doc` grounding tool. The path is validated to a same-origin `.md` page and the body is size-capped.

Parameters

Name	In	Type	Required	Description
`path`	query	string		The docs page's `.md` path (relative, no scheme/traversal), e.g. `ai-gateway.md`.

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.
`404`	Not found — `not_found`.
`429`	Rate limited — `rate_limited`.
`502`	Upstream error — `upstream_error`.

`POST /v1/assistant/escalate`

Escalate to a human

File a support ticket (a GitHub issue in the operator's repo) with the chat history, attempted actions, and account details, for a human to handle. Per-account rate-limited. Records the escalation and mirrors it to the support list; if GitHub filing is unavailable it still records + mirrors and reports `filed: false`.

Request body

Field	Type	Required
`summary`	string	
`transcript`	array of item	
`actions`	array of item	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden_key`.
`413`	Payload too large — `too_large`.
`429`	Rate limited — `rate_limited`.

`GET /v1/audit/export.csv`

Audit-log CSV export

A CSV of the account's lifecycle audit trail — account status changes, subscription events, payment events, API-key created/revoked, and SSO/SCIM security events (category `sso`: config saves, domain verifications, SCIM token mint/revoke, provisioning) — unioned into `ts,category,event,actor,detail`, newest first, capped at 10k rows (`X-Aardvark-Export-Truncated` when hit). Billing-manager + Business/Enterprise plan.

Responses

Code	Description
`200`	A CSV file (`text/csv`).
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden`, `forbidden_key`, `plan_required`.
`429`	Rate limited — `rate_limited`.

`GET /v1/authoring-analytics`

Authoring analytics

Per-type counts and the most recent authoring events over the last 90 days. Returns clean empty shapes when no rows exist.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/authoring-telemetry`

Authoring beacon

Scaffold ingest for a future authoring client. Public-key authed like the other reader beacons; `event_type` is restricted to letters/digits/'.'/'_'/'-'. Best-effort over the per-account daily cap.

Request body

Field	Type	Required
`event_type`	string	yes
`detail`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`.

`GET /v1/billing`

Plan catalog + live subscription and grant meter

The subscription-plan catalog (Free/Pro/Business/Enterprise — operator-editable data), the account's live subscription with its included-AI grant meter (dollars used, the account's own trailing burn rate, an estimated depletion date), the anomaly-breaker state, and recent subscription events. Owner/admin only, like the other billing surfaces.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden`, `forbidden_key`.

`POST /v1/billing/cancel`

Cancel the subscription (or undo a scheduled cancellation)

A Stripe-billed subscription cancels at PERIOD END (the customer keeps what they paid for; the webhook performs the downgrade at the boundary) — the scheduled end date is surfaced as the subscription's `cancel_at` until then. Pass `resume: true` to UNDO a scheduled cancellation (409 `no_pending_cancel` when none is scheduled). An operator-granted subscription downgrades immediately. The prepaid balance is never touched by a lapse.

Request body

Field	Type	Required
`resume`	boolean	

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `no_subscription`.

409	Conflict — change_pending, no_pending_cancel, not_live.
429	Rate limited — rate_limited.
502	Upstream error — cancel_failed.
503	Service unavailable — payments_unavailable.

`POST /v1/billing/change-plan`

Switch the live subscription to another plan (upgrades apply immediately; downgrades at the next boundary)

Switches the live subscription to another plan. An UPGRADE (a higher-priced plan) applies IMMEDIATELY — the Stripe price is swapped with the difference prorated onto the next invoice, and the grant/markup/seats re-snapshot now. A DOWNGRADE (a lower-priced plan) is SCHEDULED for the next billing boundary with NO current-period proration: the current plan's entitlement and billing continue unchanged until then, and the switch applies at that period's invoice.paid (the response reports the scheduled plan via `pending_plan`, and switching back before the boundary unschedules it). At most ONE off-cycle re-grant applies per grant period regardless of how many times plans are swapped (swap-proof grant minting). Operator-managed (non-Stripe) subscriptions are refused — the operator re-plans via the admin surface.

Request body

Field	Type	Required
plan	string	yes

Responses

Code	Description
200	Success.
400	Bad request — bad_plan.
401	Unauthenticated — invalid_key.
402	Payment required — change_failed.
403	Forbidden — account_closed, account_suspended, csrf, forbidden, forbidden_key.
404	Not found — no_subscription, not_found.

409	Conflict — cancel_pending, change_pending, not_live, operator_managed.
429	Rate limited — rate_limited.
503	Service unavailable — payments_unavailable.

`POST /v1/billing/overflow`

Set the included-AI overflow policy

What happens when the monthly included AI runs out: cap_hold (default — paid AI pauses; the plan price is the worst-case bill), payg_fallback (overflow draws the prepaid balance at the plan's member markup, up to the cap), or shutoff. Raising the cap past 2× the monthly grant requires the explicit acknowledgment.

Request body

Field	Type	Required
mode	string (cap_hold | payg_fallback | shutoff)	yes
cap_usd	number	
ack_2x	boolean	

Responses

Code	Description
200	Success.
400	Bad request — ack_required, bad_request.
401	Unauthenticated — invalid_key.
403	Forbidden — csrf, forbidden, forbidden_key.
404	Not found — no_subscription.
409	Conflict — not_live.
429	Rate limited — rate_limited.

`POST /v1/billing/resume-ai`

Resume paid AI after an anomaly pause

Lifts the spend-velocity circuit breaker's pause and prevents the same tripped UTC-day spend bucket from immediately re-triggering it; spend attributed to any other UTC day is evaluated normally. Works on every plan including Free — the breaker guards the pay-as-you-go tier too.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`429`	Rate limited — `rate_limited`.

`POST /v1/billing/seats`

Set the total add-on editor seat count (block + buy)

Sets the ABSOLUTE number of purchased add-on editor seats on a Stripe-billed subscription (migration 0045). The plan's `seats_included` are free; each add-on seat bills at the plan's per-seat monthly rate. Seats are block-and-buy — invites are refused with `403 seat_limit` at the limit until more are bought here, so the bill never grows without this call. A REDUCTION can't drop below the seats already occupied (members + pending invites). Operator-managed (non-Stripe) subscriptions are refused. A positive purchase on a plan that sells no add-on seats is refused with `400 seats_not_available`, but a REMOVAL (`seats: 0`) is NOT blocked by that guard even after a plan's per-seat price has been zeroed — so a customer holding a still-billing seat item can shed it rather than be trapped. (A removal is still subject to the same `400 seats_in_use` floor as any reduction — it can't drop the limit below the seats already occupied by members + pending invites — and to the suspended/closed-account guards.)

Request body

Field	Type	Required
`seats`	integer	yes

Responses

Code	Description

200	Success.
400	Bad request — `bad_request`, `seats_in_use`, `seats_not_available`.
401	Unauthenticated — `invalid_key`.
402	Payment required — `seats_failed`.
403	Forbidden — `account_closed`, `account_suspended`, `csrf`, `forbidden`, `forbidden_key`.
404	Not found — `no_subscription`, `not_found`.
409	Conflict — `operator_managed`.
429	Rate limited — `rate_limited`.
503	Service unavailable — `payments_unavailable`.

`POST /v1/billing/subscribe`

Subscribe to a plan (charges the saved card now)

Creates the Stripe subscription for a self-serve plan and grants the first month's included AI immediately. Requires a card on file (see `POST /v1/payment/checkout`). Overflow defaults to cap-and-hold at 1× the grant, so the plan price is also the worst-case bill until the customer opts into fallback.

Request body

Field	Type	Required
`interval`	string (month | year)	
`plan`	string	yes

Responses

Code	Description
200	Success.
400	Bad request — `bad_plan`, `bad_request`, `card_required`.
401	Unauthenticated — `invalid_key`.

402	Payment required — subscribe_failed.
403	Forbidden — account_closed, account_suspended, csrf, forbidden, forbidden_key.
404	Not found — not_found.
409	Conflict — account_closed, already_subscribed, card_changed, not_live.
429	Rate limited — rate_limited.
503	Service unavailable — payments_unavailable.

`POST /v1/chat/completions`

Chat completion

OpenAI-compatible chat-completions endpoint. The gateway authorizes the key, reserves the input cost against the account balance, forwards the request to the configured upstream model, and streams the response back as Server-Sent Events while metering actual spend. System-prompt handling depends on the key mode: for **public** keys (origin-gated browser/reader-widget keys) the gateway always injects its own system prompt and strips any client-supplied system message; for **secret** keys (server/CLI/automation credentials) a nonblank client-supplied `system` message is honored verbatim — so tools such as `vark author` set their own instructions — and the gateway's default prompt is used only when the request supplies no system message. The sampling fields (temperature/top_p/frequency_penalty/presence_penalty) are clamped to valid ranges.

Request body

Field	Type	Required
model	string	yes
messages	array of item	yes
tools	array of item	
plugins	array of item	
tool_choice	object	
max_tokens	integer	
stop	array of item	

`reasoning`	object	
`temperature`	number	
`top_p`	number	
`frequency_penalty`	number	
`presence_penalty`	number	

Responses

Code	Description
`200`	A `text/event-stream` of OpenAI-style completion chunks (`data: {…}` lines, terminated by `data: [DONE]`). The optional `X-Stop-Truncated: true` header signals dropped excess `stop` sequences.
`400`	Bad request — `attachments_too_large`, `bad_request`, `model_not_available`, `too_many_attachments`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`, `anomaly_paused`, `insufficient_balance`.
`403`	Forbidden — `origin_forbidden`.
`429`	Rate limited — `concurrency_limit`, `rate_limited`.
`502`	Upstream error — `upstream_error`.
`503`	Service unavailable — `pricing_unavailable`.

`POST /v1/comment`

Submit a comment

Stores a free-text reader comment, optionally tied to a page path and an island-generated survey/question id. Best-effort over the per-account daily cap. `page_url` must be a safe site-relative pathname or it is stored as null.

Request body

Field	Type	Required

`comment`	string	yes
`page_url`	string	
`survey_id`	string	
`question_id`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`.
`413`	Payload too large — `payload_too_large`.
`429`	Rate limited — `rate_limited`.

`GET /v1/conversations`

Conversation detail

Recent turns grouped into conversations, each with the reader vote and (when available) an automated analysis verdict. `filter=needs_attention` restricts to low-confidence / doc-gap / not-found turns; `verdict`/`intent`/`vote`/`q`/`tag` narrow it further.

Parameters

Name	In	Type	Required	Description
`verdict`	query	string		Restrict to a single analysis verdict.
`filter`	query	string		Set to `needs_attention` to restrict the result (default `all`).
`intent`	query	string		Restrict to a single detected intent.

`vote`	query	string		Restrict to `up`/`down` votes.
`q`	query	string		Free-text search over question/answer.
`tag`	query	string		Restrict to a custom tag.
`since`	query	integer		Epoch-ms lower bound; keeps turns with `ts` >= this (inclusive).
`until`	query	integer		Epoch-ms upper bound; keeps turns with `ts` < this (exclusive).

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`GET /v1/conversations/export.csv`

Export conversations

CSV of the conversation turns matching the same filters as `GET /v1/conversations` (every cell formula-injection-guarded). An `X-Aardvark-Export-Truncated: true` response header signals the export cap was hit.

Parameters

Name	In	Type	Required	Description
`verdict`	query	string		Restrict to a single analysis verdict.
`filter`	query	string		Set to `needs_attention` to restrict the result (default `all`).
`intent`	query	string		Restrict to a single detected intent.
`vote`	query	string		Restrict to `up`/`down` votes.
`q`	query	string		Free-text search over question/answer.

`tag`	query	string		Restrict to a custom tag.
`since`	query	integer		Epoch-ms lower bound; keeps turns with `ts` >= this (inclusive).
`until`	query	integer		Epoch-ms upper bound; keeps turns with `ts` < this (exclusive).

Responses

Code	Description
`200`	A CSV file (`text/csv`).
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`GET /v1/digest`

Get digest prefs

The account's insight-digest opt-in: whether `enabled`, the `cadence`, and when it was `last_sent_ts`.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/digest`

Set digest prefs

Opt the account owner into (or out of) a weekly/monthly emailed insights digest. Owner only. **Owner only.**

Request body

Field	Type	Required

`enabled`	boolean	yes
`cadence`	string (month | week)	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.

`POST /v1/engagement`

Engagement beacon

Public-key beacon (migration 0027): the escalation/contact link was SHOWN under a downvoted answer (`escalation_shown`) or CLICKED (`escalation_click`). Decoupled from the transcript and subject to a per-account daily row cap; drives the dashboard's support-deflection metric.

Request body

Field	Type	Required
`conversation_id`	string	yes
`kind`	string	yes
`turn`	integer	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.

403	Forbidden — origin_forbidden.
413	Payload too large — payload_too_large.
429	Rate limited — rate_limited.

GET /v1/feedback

Thumbs summary

The owner-facing READ of reader thumbs votes — the capture POST is POST /v1/feedback. Returns up/down counts, the satisfaction rate, and recent rated answers. Dashboard-authed (session OR secret key); a public key is rejected at the door.

Responses

Code	Description
200	Success.
401	Unauthenticated — invalid_key.
403	Forbidden — forbidden_key.

POST /v1/feedback

Rate an answer

Stores a reader's vote on a specific answer turn. Best-effort: over the per-account daily cap the vote is silently dropped but still returns 200. Deduplicated on (account, conversation_id, turn).

Request body

Field	Type	Required
vote	string (up | down)	yes
conversation_id	string	yes
question	string	
turn	integer	

Responses

Code	Description
200	Success.
400	Bad request — `bad_request`.
401	Unauthenticated — `invalid_key`, `revoked_key`.
402	Payment required — `account_inactive`.
403	Forbidden — `origin_forbidden`.
413	Payload too large — `payload_too_large`.
429	Rate limited — `rate_limited`.

`GET /v1/feedback/summary`

Feedback summary

One cross-stream snapshot of the account's reader feedback over a recent 30-day window — thumbs, page ratings, survey responses, and conversation sentiment — powering the metric tiles at the top of the dashboard Feedback panel. Each sub-stream degrades to zeros if its backing table isn't present yet.

Responses

Code	Description
200	Success.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `forbidden_key`.

`GET /v1/github/capabilities`

List capabilities

The static catalog of AI authoring capabilities (e.g. keyword and description generation, the style-guide pass) that can be enabled to run on a connected repo. Readable by any authenticated dashboard user.

Responses

Code	Description

`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.
`503`	Service unavailable — `github_unconfigured`.

`POST /v1/github/configure`

Configure repo

Declarative per-repo automation config: for each capability set whether it's enabled, its schedule (`cron`, floored to at most once per hour), run-on-push, base branch, and project dir. The `styleguide` capability additionally REQUIRES `rulesets`: a non-empty, duplicate-free array of style ruleset ids (see `style_rulesets` on `GET /v1/github/connection`) in precedence order — index 0 wins conflicts; `rulesets` is rejected on every other capability. A pure DB apply — no GitHub writes. Owner only. **Owner only.**

Request body

Field	Type	Required
`repo`	string	yes
`automations`	array of object	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `invalid_cron`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.
`409`	Conflict — `install_mismatch`, `install_missing`, `install_stale`, `install_suspended`.
`503`	Service unavailable — `github_unconfigured`.

`POST /v1/github/connect`

Connect a repo

Mints a single-use install `state` and returns the GitHub App install URL carrying it. The browser follows that URL; GitHub echoes `state` back to the setup callback, which binds the installation to this account. Owner only. **Owner only.**

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`503`	Service unavailable — `github_unconfigured`.

`GET /v1/github/connection`

Connection status

Everything the dashboard's Docs Quality Checks page needs: whether the integration is configured, the install-URL slug, the account's installations, and each connected repo with its per-capability automation config.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.
`503`	Service unavailable — `github_unconfigured`.

`POST /v1/github/disconnect`

Disconnect a repo

Stops all automations for a repo by dropping their config rows in one atomic delete. Leaves the app installed (uninstall happens in GitHub) and the repo listed so it can be reconfigured. Owner only. **Owner only.**

Request body

Field	Type	Required
`repo`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.
`409`	Conflict — `install_mismatch`.
`503`	Service unavailable — `github_unconfigured`.

`POST /v1/github/run`

Run now

Dispatches a configured capability immediately on the central runner. The run is recorded as `queued` and advanced by the runner's claim/complete callbacks. A balance or runner-config problem returns 402/503; an already-running capability returns 409; a `styleguide` automation with no stored ruleset selection (a legacy row from before rulesets existed) returns 409 `rulesets_required` until it's re-saved with rulesets picked. Owner only. **Owner only.**

Request body

Field	Type	Required
`repo`	string	yes
`capability`	string	yes
`path`	string	

Responses

Code	Description
200	Success.
400	Bad request — bad_request.
401	Unauthenticated — invalid_key.
402	Payment required — insufficient_balance.
403	Forbidden — csrf, forbidden, forbidden_key.
404	Not found — not_found.
409	Conflict — account_run_cap, already_running, install_mismatch, install_missing, install_stale, install_suspended, not_configured, rulesets_required.
502	Upstream error — github_error.
503	Service unavailable — github_unconfigured, runner_unconfigured.

`POST /v1/github/run/cancel`

Cancel run

Cancels the active queued or in-progress run for a repo/capability pair, revokes any already-minted ephemeral inference key, and best-effort cancels the backing GitHub Actions run. Idempotent: idle pairs return `cancelled: false`. **Owner only.**

Request body

Field	Type	Required
repo	string	yes
capability	string	yes

Responses

Code	Description
200	Success.
400	Bad request — bad_request.

401	Unauthenticated — invalid_key.
403	Forbidden — csrf, forbidden, forbidden_key.
503	Service unavailable — github_unconfigured.

`POST /v1/github/runner/claim`

Claim a run

Central-runner callback: the runner claims a queued run and receives an ephemeral inference key plus a repo-scoped GitHub token after the claim CAS wins. An exact same-Actions-run deploy retry is idempotent: it keeps the already-bound one-time inference key, omits that secret from the response, and reissues a least-privilege repo token. Authed by the runner credential (a GitHub Actions OIDC JWT, verified against GitHub's JWKS), never a dashboard session.

Request body

Field	Type	Required
run_id	string	yes
github_run_id	integer	yes

Responses

Code	Description
200	Success.
400	Bad request — bad_request.
401	Unauthenticated — unauthorized.
404	Not found — not_found.
409	Conflict — already_claimed, bad_run, install_mismatch, install_stale, install_suspended, rulesets_required, run_binding_mismatch, site_disabled.
500	Server error — claim_failed, key_bind_failed, key_mint_failed.
502	Upstream error — repo_token_failed.
503	Service unavailable — binding_pending.

`POST /v1/github/runner/complete`

Complete a run

Central-runner callback: reports a finished run. Revokes the run's ephemeral key (stops spend), meters the compute, and records the conclusion plus the PR it opened. Idempotent; a late callback can top up a $0-sealed run. Authed by the runner credential.

Request body

Field	Type	Required
`github_run_id`	integer	yes
`run_id`	string	yes
`conclusion`	string	
`billable_ms`	number	
`pr_url`	string (uri)	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `unauthorized`.
`404`	Not found — `not_found`.
`409`	Conflict — `not_in_progress`, `run_mismatch`.
`500`	Server error — `deploy_seal_failed`, `finalize_failed`.

`PUT /v1/github/runner/deploy-file`

Upload a deploy file

Central-runner callback for managed-hosting deploys: streams ONE built site file per request (raw body) into the deploy's immutable R2 prefix. Query params: `run` (the gateway run UUID), `gh_run` (the Actions run id, bound to the OIDC token), and `path` (urlencoded site-relative file path — traversal shapes are rejected). Enforces the per-file/per-deploy budgets (25MB/file, 10k files,

500MB total) and stamps the content type from the shared extension map. Authed by the runner credential, never a dashboard session.

Parameters

Name	In	Type	Required	Description
`gh_run`	query	integer	yes	The positive GitHub Actions run id bound to the runner OIDC token.
`run`	query	string	yes	The gateway run UUID for this deploy.
`path`	query	string	yes	URL-encoded site-relative output path.
`Content-Length`	header	integer	yes	Exact raw-body size in bytes; required for the per-file and per-deploy upload budgets.

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_path`.
`401`	Unauthenticated — `unauthorized`.
`411`	Error — `length_required`.
`413`	Payload too large — `deploy_too_large`, `file_too_large`.
`503`	Service unavailable — `hosting_unconfigured`.

`POST /v1/github/runner/deploy-finalize`

Finalize a deploy

Central-runner callback for managed-hosting deploys: every file is uploaded, so flip the deploy LIVE — supersede the previous live deploy for the (site, branch) and write the `routes/<hostname>` pointer object(s) the sites-worker serves from. Query param `run` carries the gateway run UUID. Idempotent (a retry re-writes the pointers). Compute metering still happens on the normal complete callback. Authed by the runner credential.

Parameters

Name	In	Type	Required	Description

`run`	query	string	yes	

Request body

Field	Type	Required
`github_run_id`	integer	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `unauthorized`.
`404`	Not found — `not_found`.
`409`	Conflict — `bad_run`, `deploy_not_open`, `empty_deploy`, `not_in_progress`, `run_mismatch`, `site_disabled`.
`500`	Server error — `attach_reconcile_cleanup_failed`, `attach_reconcile_failed`, `detach_cleanup_failed`, `pointer_write_failed`, `takedown_cleanup_failed`.
`503`	Service unavailable — `hosting_unconfigured`.

`GET /v1/github/runs`

List runs

Run history for a connected repo, newest first, optionally filtered by `capability`. Each row carries status, conclusion, the PR the run opened, and the metered compute cost.

Parameters

Name	In	Type	Required	Description
`repo`	query	string	yes	Repo `owner/name` to list runs for (required).
`capability`	query	string		Restrict to one capability's runs.
`limit`	query	integer		Max rows to return (default 50, capped at 200).

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.
`404`	Not found — `not_found`.
`503`	Service unavailable — `github_unconfigured`.

`DELETE /v1/github/runs/{run_id}`

Clear a run history entry

Deletes one terminal GitHub automation run from the account's dashboard history. Queued and in-progress runs must be cancelled first because they still gate dispatch and key cleanup. **Owner only.**

Parameters

Name	In	Type	Required	Description
`run_id`	path	string	yes	Stable run UUID from `GET /v1/github/runs`.

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.
`409`	Conflict — `active_run`, `compute_pending`.

`503`	Service unavailable — `github_unconfigured`.

`GET /v1/github/runs/{run_id}/logs`

Read run logs

Returns the latest GitHub Actions job log text for one Docs Quality Check run. The dashboard polls this endpoint while a row is expanded so queued and in-progress runs can surface logs as soon as GitHub publishes them.

Parameters

Name	In	Type	Required	Description
`run_id`	path	string	yes	Stable run UUID from `GET /v1/github/runs`.

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.
`404`	Not found — `not_found`.
`429`	Rate limited — `rate_limited`.
`502`	Upstream error — `github_logs_unavailable`.
`503`	Service unavailable — `github_unconfigured`, `runner_unconfigured`.

`GET /v1/github/setup`

Install callback

GitHub redirects the browser here after the app is installed or updated. The single-use `state` (minted at connect) recovers which account initiated, an identity check proves the user administers the installation, then it binds the install, syncs repos, and 302-redirects back to the dashboard. Authed by `state`, not a session.

Parameters

Name	In	Type	Required	Description
`state`	query	string		Single-use install state minted at connect; the credential that recovers and binds the initiating account.
`installation_id`	query	integer		GitHub's numeric id for the new/updated installation.
`code`	query	string		OAuth code GitHub appends so the gateway can prove the user administers this installation.

Responses

Code	Description
`302`	Redirect (HTTP 302) to `/dashboard/github?connected=1` on success, or `?error=<code>` otherwise.
`503`	Service unavailable — `github_misconfigured`, `github_unavailable`.

`POST /v1/github/webhook`

GitHub webhook

GitHub-to-server webhook authenticated by the `X-Hub-Signature-256` HMAC over the raw body. Handles installation lifecycle, repo-selection changes, pushes (run-on-push automations), workflow runs (a backstop for the runner's complete callback), and PRs (PR-url capture). Acks 200 once verified; unrecognized event types are acked and ignored.

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `invalid_signature`.
`500`	Server error — `internal_error`.
`503`	Service unavailable — `webhook_unconfigured`.

`POST /v1/insight-status`

Set triage status

Mark a clustered insight (by `label`) as e.g. triaged/resolved with an optional note — drives the dashboard's coverage-gap workflow.

Request body

Field	Type	Required
`label`	string	yes
`status`	string	yes
`note`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden_key`.

`GET /v1/insights`

Clustered insights

The cron-clustered analytics surface: live metrics plus Top Questions and Coverage Gaps for the selected period, with prior-period deltas when available.

Parameters

Name	In	Type	Required	Description
`days`	query	integer		Rolling window in days.
`period_kind`	query	string		Named period bucket (e.g. `week` / `month`).
`period_start`	query	integer		Period start, epoch ms.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/insights/assistant`

Ask the assistant

Natural-language Q&A over the account's analytics. Metered against the account balance (it calls the model), so it can return 402/429/502/503 like the chat endpoint.

Request body

Field	Type	Required
`question`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`402`	Payment required — `insufficient_balance`.
`403`	Forbidden — `csrf`, `forbidden_key`.
`429`	Rate limited — `rate_limited`.
`502`	Upstream error — `upstream_error`.
`503`	Service unavailable — `pricing_unavailable`, `unavailable`.

`GET /v1/keys`

List API keys

Returns metadata for every key on the account (prefix, mode, label, allowed origins, revoked flag) — never the hash or secret value.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/keys`

Mint a new API key

Creates a public or secret key. The full key value is returned ONCE. Secret keys are owner-only and capped at one active per account (rotate to replace). Public keys may be origin-scoped.

Request body

Field	Type	Required
`mode`	string (public | secret)	
`label`	string	
`allowed_origins`	array of item	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`, `member_inactive`.
`409`	Conflict — `secret_key_exists`.

`POST /v1/keys/{id}`

Rename / rescope key

Updates a key's label and/or a public key's allowed origins. At least one field is required; an omitted field is left unchanged. Non-owners may only update keys they created.

Parameters

Name	In	Type	Required	Description
`id`	path	string	yes	Identifier of the key to update.

Request body

Field	Type	Required
`label`	string	
`allowed_origins`	array of item	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.

`POST /v1/keys/{id}/revoke`

Revoke a public key

Permanently revokes a PUBLIC key. Secret keys cannot be bare-revoked (rotate them instead) to avoid locking the owner out.

Parameters

Name	In	Type	Required	Description

`id`	path	string	yes	Identifier of the public key to revoke.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.

`POST /v1/keys/{id}/rotate`

Hard-rotate a key

Revokes the key and mints a same-mode replacement atomically. The new value is shown once. Non-owners may only rotate keys they created; a demoted member cannot rotate the account secret key.

Parameters

Name	In	Type	Required	Description
`id`	path	string	yes	Identifier of the key to rotate.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`, `member_inactive`.
`404`	Not found — `not_found`.
`409`	Conflict — `key_not_rotatable`.

`GET /v1/mcp-analytics`

MCP tool analytics

Top tools, recent calls, and a bucketed time series over a window of `days` (default 7, max 90). Buckets are hourly for ≤2 days, daily otherwise.

Parameters

Name	In	Type	Required	Description
`days`	query	integer		Window in days (1–90).

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/mcp-telemetry`

MCP tool-call beacon

The docs MCP server posts one beacon per tool call. A short args summary is either supplied (`args_summary`) or derived from `args`. Server-side callers (no Origin header) bypass the origin allowlist; browsers must pass it. Best-effort over the per-account daily cap.

Request body

Field	Type	Required
`tool`	string	yes
`args_summary`	string	
`args`	object	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.

`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`.

`GET /v1/page-ratings`

Page ratings

The dashboard READ of reader star ratings: the overall average, total count, and a per-page breakdown. Dashboard-authed like `/v1/insights`.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/payment/autotopup`

Auto top-up settings

Sets the auto-top-up on/off state, charge amount, and low-balance trigger. At least one field is required; the trigger must be below the charge amount. Enabling requires a saved card and an amount. Owner/admin only.

Request body

Field	Type	Required
`enabled`	boolean	
`amount_usd`	number	
`low_balance_usd`	number	

Responses

Code	Description

`200`	Success.
`400`	Bad request — `bad_request`, `no_card`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.
`429`	Rate limited — `rate_limited`.
`503`	Service unavailable — `payments_unavailable`.

`POST /v1/payment/checkout`

Start card setup

Creates (if needed) a Stripe customer and a hosted SetupIntent Checkout Session, returning the redirect URL. No body required. Owner/admin only.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.
`429`	Rate limited — `rate_limited`.
`502`	Upstream error — `stripe_error`.
`503`	Service unavailable — `payments_unavailable`.

`POST /v1/payment/method`

Finalize saved card

Server-side re-reads the completed Checkout Session, verifies it belongs to this account, and stores the card on file. Idempotent on a given session_id. Owner/admin only.

Request body

Field	Type	Required
`session_id`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `card_setup_incomplete`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`429`	Rate limited — `rate_limited`.
`500`	Server error — `db_error`.
`502`	Upstream error — `stripe_error`.
`503`	Service unavailable — `payments_unavailable`.

`DELETE /v1/payment/method`

Remove saved card

Clears the card on file (also disabling auto-top-up) and best-effort detaches it at Stripe. Idempotent; allowed even when card payments are disabled. Owner/admin only.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`429`	Rate limited — `rate_limited`.

500	Server error — db_error.
502	Upstream error — stripe_error.
503	Service unavailable — payments_unavailable.

`POST /v1/payment/topup-now`

Charge saved card

Immediately charges the saved card for amount_usd and credits the balance. Use a client idempotency_key to make retries safe. A pending charge returns { pending: true } and is credited by the webhook. Set reactivate: true to lift a self-serviceable (cost-unavailable) suspension before charging, so one call reactivates and tops up. Owner/admin only.

Request body

Field	Type	Required
reactivate	boolean	
amount_usd	number	yes
idempotency_key	string	

Responses

Code	Description
200	Success.
400	Bad request — bad_request, no_card, payment_error.
401	Unauthenticated — invalid_key.
402	Payment required — card_declined.
403	Forbidden — account_closed, account_not_active, csrf, forbidden, forbidden_key, operator_hold.
404	Not found — not_found.
409	Conflict — account_merge_in_progress, topup_in_progress.
429	Rate limited — rate_limited.

`503`	Service unavailable — `payments_unavailable`, `service_unavailable`.

`POST /v1/rating`

Submit a rating

Public-key capture of a reader star rating for a page (migration 0036), optionally with a comment. Best-effort over the per-account daily cap.

Request body

Field	Type	Required
`rating`	string	yes
`comment`	string	
`page_url`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`.
`413`	Payload too large — `payload_too_large`.
`429`	Rate limited — `rate_limited`.

`GET /v1/reach-analytics`

Content Reach analytics

Aggregate Content Reach over a window of `days` (default 30, max 90): page-view and session totals, average foreground-visible time, average sections reached, totals for each of the five tracked actions, and a per-page table with reach completion. Plan-gated to Business/Enterprise (`403 plan_required`). Averages are clamped means — the clamp is applied at ingest, so a tab left

open cannot drag them. Everything is aggregate: there is no endpoint returning an individual reader's path, because there is no durable visitor identity to build one from.

Parameters

Name	In	Type	Required	Description
`days`	query	integer		Window in days (1–90; default 30).

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`, `plan_required`.

`POST /v1/reach-event`

Content Reach beacon

Public-key beacon (migration 0091): one page-view's worth of SECTION-level visibility and interaction evidence, posted by `/_aardvark/reach.js`. Records which of a page's own semantic sections (heading slugs, `target`-tag ids, and the synthetic `__intro`) became at least 50% visible for at least 1 second, how long each was the section under the reader's trigger line, and counts for a closed five-value action taxonomy. It carries no coordinates, scroll offsets, DOM snapshots, element text, or query text, and no identifier durable beyond one browser tab — so it is visibility and interaction evidence, never a measure of gaze, comprehension, or attention. Unlike the other reader-write endpoints this one is PLAN-GATED to Business/Enterprise (`403 plan_required`); the gateway is authoritative because a static build holds only the public key and cannot read the account's plan. Repeat posts of the same `(session_id, session_seq)` are expected — the client flushes on tab-hide and again on pagehide — and are idempotent, only ever moving stored values forward.

Request body

Field	Type	Required
`session_id`	string	yes
`session_seq`	integer	yes
`page_url`	string	yes

`visible_ms`	integer	
`sections_total`	integer	
`sections`	array of object	
`catalog`	array of object	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`, `plan_required`.
`413`	Payload too large — `payload_too_large`.
`429`	Rate limited — `rate_limited`.

`GET /v1/reach-page`

Content Reach for one page

Per-page Content Reach drill-in — the data behind the dashboard's section heat strip. Returns one row per section in document order with its reach rate, median foreground dwell, and action counts, plus the distribution of last-reached sections. Reach counts and action totals are exact over the window; the dwell statistic is a MEDIAN over a bounded sample, so one long-parked tab cannot skew it. Plan-gated to Business/Enterprise (`403 plan_required`).

Parameters

Name	In	Type	Required	Description
`path`	query	string	yes	Site path of the page to inspect, normalized the same way the ingest normalized it.
`days`	query	integer		Window in days (1–90; default 30).

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`, `plan_required`.

`POST /v1/reactivate`

Reactivate account

Lifts the gateway's automatic cost-unavailable suspension. Operator-set holds and closed accounts cannot be reactivated here. Idempotent. Owner/admin only.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `account_closed`, `csrf`, `forbidden`, `forbidden_key`, `operator_hold`.
`404`	Not found — `not_found`.
`503`	Service unavailable — `service_unavailable`.

`GET /v1/search-analytics`

Search analytics

On-site search analytics over a window of `days` (default 30, max 90): top searches, zero-result queries, click-through, average clicked position, searches-per-session, no-click rate, Ask-AI escalation rate, top clicked pages, a language split, and a bucketed volume series (hourly for ≤2 days, daily otherwise). The response also includes the latest durable calendar-period search-theme/content-gap snapshot, which is independent of the requested live-event window.

Parameters

Name	In	Type	Required	Description

`days`	query	integer		Window in days (1–90; default 30).

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/search-event`

Search activity beacon

Public-key batch beacon (migration 0040): the header search box reports settled queries, result clicks, and Ask-AI escalations. Body is `{ events: [...] }`, each event a `query`, `click`, or `ask_ai`. Subject to a per-account daily row cap and dedup; only AI-enabled sites emit. Powers the dashboard's Search Analytics.

Request body

Field	Type	Required
`events`	array of item	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`.
`413`	Payload too large — `payload_too_large`.
`429`	Rate limited — `rate_limited`.

`GET /v1/sentiment`

Reader sentiment

The positive/neutral/negative distribution of analyzed conversations over a recent window, plus recent example messages, for the reader-feedback panel. Dashboard-authed like `/v1/insights`.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`GET /v1/sites`

Get the hosted site

The account's managed-hosting site (v1: at most one) with its recent deploys and the URLs it serves — the production URL, the branch-preview pattern, and any custom hostname. NOT plan-gated: reads (and the DELETE takedown paths) stay available even after a plan lapse so a downgraded owner can still see and take down their site — the payload's `hosting_plan_allowed` flag reports whether NEW deploys/attaches are allowed (only POST create/deploy/attach return 403 `plan_required`). 503 when hosting isn't configured on this gateway. Owner/admin only.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden`, `forbidden_key`.
`503`	Service unavailable — `hosting_unconfigured`.

`POST /v1/sites`

Create the hosted site

Creates the account's hosted site: a dns-safe slug served at `https://<slug>.<base domain>` (in custom-domains-only mode the slug backs preview URLs and production goes live on a custom domain attached afterward — the site is URL-less until then), bound to a repo from the account's

GitHub connection. Every push to the repo then builds and deploys automatically (in PSL mode the production branch goes live at `<slug>.<base>`; in custom-domains-only mode production goes live on the attached custom domain; other branches get preview hostnames in both modes). One site per account in v1 (409 `site_exists`); a taken slug returns 409 `slug_taken`. Plan-gated to Pro and above. Owner/admin only.

Request body

Field	Type	Required
`slug`	string	yes
`repo_id`	integer	yes
`production_branch`	string	
`project_dir`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `bad_slug`, `reserved_slug`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `account_inactive`, `csrf`, `forbidden`, `forbidden_key`, `plan_required`.
`404`	Not found — `repo_not_connected`.
`409`	Conflict — `site_exists`, `slug_taken`.
`429`	Rate limited — `rate_limited`.
`503`	Service unavailable — `hosting_unconfigured`, `stale_pointer_clear_failed`.

`DELETE /v1/sites/{site_id}`

Delete the hosted site

Disables the site and stops serving it: the hostname route pointers are removed and any custom hostname is detached best-effort. Deployed content objects are kept in R2 (cheap, and no destructive bulk-delete path). Owner/admin only.

Parameters

Name	In	Type	Required	Description
`site_id`	path	string	yes	The site UUID from `GET /v1/sites`.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`429`	Rate limited — `rate_limited`.
`500`	Server error — `delete_failed`.
`502`	Upstream error — `cf_error`.
`503`	Service unavailable — `hosting_unconfigured`, `takedown_incomplete`.

`POST /v1/sites/{site_id}/deploy`

Deploy now

Manually enqueues a build-and-deploy run for a branch (default: the production branch). The deploy executes on the central runner and bills compute like any automation run; an unpayable balance returns 402, an in-flight deploy for the same branch returns 409 `deploy_in_flight`. Owner/admin only.

Parameters

Name	In	Type	Required	Description
`site_id`	path	string	yes	The site UUID from `GET /v1/sites`.

Request body

Field	Type	Required
`branch`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`402`	Payment required — `insufficient_balance`.
`403`	Forbidden — `account_inactive`, `csrf`, `forbidden`, `forbidden_key`, `plan_required`.
`409`	Conflict — `account_run_cap`, `deploy_in_flight`, `install_mismatch`, `install_missing`, `install_stale`, `install_suspended`, `site_disabled`.
`429`	Rate limited — `rate_limited`.
`502`	Upstream error — `dispatch_failed`.
`503`	Service unavailable — `hosting_unconfigured`, `runner_unconfigured`.

`GET /v1/sites/{site_id}/domain`

Custom-domain status

Polls the attached custom hostname's validation + certificate status from Cloudflare for SaaS. 404 `no_domain` when the site has no custom domain. Owner/admin only.

Parameters

Name	In	Type	Required	Description
`site_id`	path	string	yes	The site UUID from `GET /v1/sites`.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden`, `forbidden_key`.

404	Not found — no_domain.
429	Rate limited — rate_limited.
502	Upstream error — cf_error.
503	Service unavailable — custom_domains_unavailable, hosting_unconfigured.

`POST /v1/sites/{site_id}/domain`

Attach a custom domain

Attaches a customer-owned hostname to the site via Cloudflare for SaaS. REQUIRES a confirmed DNS ownership proof for that exact hostname first — call `POST /v1/sites/{site_id}/domain/verify`, publish the TXT record it returns, and call it again until it reports `verified: true`; until then this returns 403 `domain_verification_required` (a proof also expires, after which it must be re-confirmed). Returns the validation records the customer must publish plus the CNAME target; poll `GET /v1/sites/{site_id}/domain` for status. 503 `custom_domains_unavailable` unless the operator configured CF_API_TOKEN + CF_ZONE_ID — in PSL / shared-subdomain mode production on `<slug>.<base>` keeps working regardless, but in custom-domains-only mode an attached custom domain is the ONLY production path, so attach being unavailable means the site has no production URL until CF for SaaS is configured. Owner/admin only.

Parameters

Name	In	Type	Required	Description
site_id	path	string	yes	The site UUID from `GET /v1/sites`.

Request body

Field	Type	Required
hostname	string	yes

Responses

Code	Description
200	Success.
400	Bad request — bad_hostname.
401	Unauthenticated — invalid_key.

403	Forbidden — account_inactive, csrf, domain_verification_required, forbidden, forbidden_key, plan_required.
404	Not found — not_found.
409	Conflict — domain_exists, hostname_cooling_down, hostname_taken, site_disabled.
429	Rate limited — rate_limited.
500	Server error — domain_cleanup_pending, domain_pointer_failed.
502	Upstream error — cf_error.
503	Service unavailable — custom_domains_unavailable, hosting_unconfigured.

`DELETE /v1/sites/{site_id}/domain`

Detach the custom domain

Detaches the custom hostname: removes its route pointer (serving stops immediately) and clears it from the site. When Cloudflare is still configured it also releases the CF-for-SaaS hostname (a real CF failure returns a retryable 5xx); if the operator removed/rotated CF creds after the domain was attached, detach still succeeds and the CF object is left as a warn-logged orphan for manual reaping rather than stranding the customer. NOT gated on Cloudflare being configured (unlike attach/status). In PSL mode the `<slug>.<base>` subdomain URL keeps serving; in custom-domains-only mode this was the site's only production URL, so production goes offline until another custom domain is attached. Owner/admin only.

Parameters

Name	In	Type	Required	Description
site_id	path	string	yes	The site UUID from `GET /v1/sites`.

Responses

Code	Description
200	Success.
401	Unauthenticated — invalid_key.
403	Forbidden — csrf, forbidden, forbidden_key.
404	Not found — no_domain.

429	Rate limited — `rate_limited`.
500	Server error — `detach_failed`.
502	Upstream error — `cf_error`.
503	Service unavailable — `hosting_unconfigured`.

`POST /v1/sites/{site_id}/domain/verify`

Prove custom-domain ownership

Mints (or re-reads) the DNS TXT challenge that `POST /v1/sites/{site_id}/domain` requires, and confirms it against live DNS in the same call. Call it once to learn the record, publish that record in the hostname's zone, then call it again — it returns `verified: true` once the record resolves, and the attach is permitted from that point. A not-yet-published record is NOT an error: the call returns 200 with `verified: false` so a client can poll while DNS propagates. The token is stable per (account, hostname), so re-calling never invalidates a record you already published. The challenge is bound to the EXACT hostname being attached (`_aardvark-verify.docs.acme.com` for `docs.acme.com`), never to a parent domain. Proofs expire and must be re-confirmed. Owner/admin only.

Parameters

Name	In	Type	Required	Description
`site_id`	path	string	yes	The site UUID from `GET /v1/sites`.

Request body

Field	Type	Required
`hostname`	string	yes

Responses

Code	Description
200	Success.
400	Bad request — `bad_hostname`.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `account_inactive`, `csrf`, `forbidden`, `forbidden_key`, `plan_required`.

404	Not found — not_found.
429	Rate limited — rate_limited.
503	Service unavailable — hosting_unconfigured.

`GET /v1/sso/config`

Get SSO config

The account's identity-provider configuration with the OIDC client secret masked, plus saml_enabled (whether this gateway exposes the SAML option). Dashboard-authed and owner-only (an admin/billing-manager is refused with 403 forbidden). The read is NOT plan-gated: a downgraded or lapsed account still receives 200 with plan_gated: true (anti-lockout) so it can view — and DELETE — an existing config; a GET never returns 403 plan_required. **Owner only.**

Responses

Code	Description
200	Success.
401	Unauthenticated — invalid_key.
403	Forbidden — forbidden, forbidden_key.

`POST /v1/sso/config`

Upsert SSO config

Configure the account's IdP. For type: "oidc" the gateway fetches the discovery document server-side (HTTPS-only, private hosts refused) and captures the issuer + endpoints. For type: "saml" (only when the gateway has SAML enabled) it stores the pinned entity id, SSO URL, and signing certificate. Which provider fields are mandatory depends on type: only type and email_domain are required of every request, and the body is a oneOf over one complete schema per type, selected by the type discriminator, each requiring the extra fields that type demands. Omitting client_secret/idp_cert_pem on an edit preserves the stored value — which is why neither branch REQUIRES them even though a FIRST save of that type does. Owner-only, Business/Enterprise plan (403 plan_required otherwise). **Owner only.**

Request body

Exactly one of these shapes:

oidc

Field	Type	Required

`type`	string (oidc)	yes
`email_domain`	string	yes
`enabled`	boolean	
`enforce_sso`	boolean	
`jit_provisioning`	boolean	
`discovery_url`	string	yes
`client_id`	string	yes
`client_secret`	string	
`idp_entity_id`	string	
`idp_sso_url`	string	
`idp_cert_pem`	string	

saml

Field	Type	Required
`type`	string (saml)	yes
`email_domain`	string	yes
`enabled`	boolean	
`enforce_sso`	boolean	
`jit_provisioning`	boolean	
`discovery_url`	string	
`client_id`	string	
`client_secret`	string	
`idp_entity_id`	string	yes

`idp_sso_url`	string	yes
`idp_cert_pem`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `discovery_failed`, `saml_disabled`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `domain_not_verified`, `forbidden`, `forbidden_key`, `plan_required`.
`409`	Conflict — `config_conflict`, `domain_already_configured`, `secret_undecryptable`.
`429`	Rate limited — `rate_limited`.
`503`	Service unavailable — `encryption_unavailable`.

`DELETE /v1/sso/config`

Delete SSO config

Remove the account's IdP configuration (also clears `enforce_sso`). Owner-only. Unlike the POST upsert, DELETE is PLAN-EXEMPT (anti-lockout): a downgraded or lapsed account can always tear down its IdP config, so a lapsed plan can never leave a team locked into an `enforce_sso` config it can't remove. **Owner only.**

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.

`GET /v1/sso/domain`

List verified domains

List the account's SSO domain-ownership records and their verification status (migration 0056). A domain must be verified before an IdP can be ENABLED for it. UNVERIFIED (including EXPIRED — proofs age out) rows also echo the DNS TXT challenge (`record_name`/`record_type`/`record_value`, same as the POST) so the record survives a dashboard reload; currently-verified rows omit it. Owner-only. PLAN-EXEMPT (anti-lockout teardown): listing stays available after a downgrade — alongside the domain DELETE — so a lapsed-plan owner can still see and remove a domain proof; only ADDING/verifying a domain (and configuring an IdP) requires the Business/Enterprise plan. **Owner only.**

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden`, `forbidden_key`.

`POST /v1/sso/domain`

Request a domain challenge

Mint (or return the existing) DNS TXT challenge the account must publish to prove it controls `domain` before SSO can be enabled for it (migration 0056). The token is stable across re-requests. Owner-only, Business/Enterprise plan. **Owner only.**

Request body

Field	Type	Required
`domain`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`, `plan_required`.
`429`	Rate limited — `rate_limited`.

`DELETE /v1/sso/domain`

Remove a domain

Remove a domain-ownership record by `domain` (query parameter). Owner-only. PLAN-EXEMPT (anti-lockout teardown): a downgraded or lapsed account can always remove a domain proof — alongside the domain GET listing — so it's never locked into an enforced config it can't dismantle. Removing the proof also disables any IdP still enabled for that domain, so the `enabled ⇒ verified` invariant holds. **Owner only.**

Parameters

Name	In	Type	Required	Description
`domain`	query	string	yes	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.

`POST /v1/sso/domain/verify`

Verify a domain

Resolve the DNS TXT challenge for `domain` and, if the issued token is present, mark the domain verified (migration 0056). Idempotent; a missing/mismatched record returns `400 verification_failed` (retry after DNS propagates). Owner-only, Business/Enterprise plan. **Owner only.**

Request body

Field	Type	Required
`domain`	string	yes

Responses

Code	Description

`200`	Success.
`400`	Bad request — `bad_request`, `verification_failed`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`, `plan_required`.
`404`	Not found — `not_found`.
`429`	Rate limited — `rate_limited`.

`GET /v1/sso/scim-token`

List SCIM tokens

List the account's active SCIM tokens (id, label, created, last used) — never the token value. Owner-only. PLAN-EXEMPT (anti-lockout teardown): a downgraded account can still discover a live provisioning credential in order to revoke it; only minting stays plan-gated. **Owner only.**

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden`, `forbidden_key`.

`POST /v1/sso/scim-token`

Mint a SCIM token

Mint a bearer token for the SCIM v2 provisioning surface (`/scim/v2/*`). The raw `aardvark_scim_…` value is returned ONCE; only its SHA-256 hash is stored. Owner-only, Business/Enterprise plan (`403 plan_required` otherwise) — minting a new provisioning credential is a plan capability. (Listing and revoking existing tokens, below, stay plan-exempt for anti-lockout.) **Owner only.**

Request body

Field	Type	Required
`label`	string	

Responses

Code	Description
200	Success.
401	Unauthenticated — invalid_key.
403	Forbidden — csrf, forbidden, forbidden_key, plan_required.
429	Rate limited — rate_limited.

`DELETE /v1/sso/scim-token`

Revoke a SCIM token

Revoke a SCIM token by `id` (query parameter). Owner-only. PLAN-EXEMPT (anti-lockout teardown): a downgraded account can always revoke a live — possibly compromised — provisioning credential without upgrading. **Owner only.**

Parameters

Name	In	Type	Required	Description
id	query	string	yes	

Responses

Code	Description
200	Success.
400	Bad request — bad_request.
401	Unauthenticated — invalid_key.
403	Forbidden — csrf, forbidden, forbidden_key.

`POST /v1/stripe/webhook`

Stripe webhook

Stripe-to-server webhook authenticated by the `Stripe-Signature` header (HMAC over the raw body). Handles `payment_intent.succeeded` (durable credit) and `payment_intent.payment_failed` (audit + auto-top-up streak), plus the subscription lifecycle (migration 0044): `customer.subscription.created`/`customer.subscription.updated` (snapshot sync, plan reconcile, lost-insert heal), `customer.subscription.deleted` (downgrade to Free), `invoice.paid` (billing history, past-due recovery, scheduled-downgrade apply at the boundary), and

`invoice.payment_failed` (dunning + past_due grace clock). Acks 200 once a delivery is handled or irrelevant; deliberately returns 500 on transient read/write failures so Stripe redelivers instead of losing a money event.

Responses

Code	Description
`200`	Success.
`400`	Bad request — `invalid_signature`.
`500`	Server error — `credit_failed`, `db_error`, `stripe_error`.
`503`	Service unavailable — `webhook_unconfigured`.

`GET /v1/support`

List support tickets

Lists tickets for the account. Plain members see only their own; owners/admins/secret-key callers see the whole account. Capped at 100 (newest first).

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/support`

Open a support ticket

Files a ticket to the gateway operator. Fail-hard over the per-account daily cap (429). Sends a best-effort operator email; the stored row is the source of truth.

Request body

Field	Type	Required
`category`	string (bug | feature | question | other)	yes

`message`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden_key`.
`429`	Rate limited — `rate_limited`.

`POST /v1/survey`

Submit survey answer

Public-key capture of a structured reader survey answer (migration 0037) — a single/multi choice or a rating — that the Survey island previously sent only to analytics. Best-effort over the per-account daily cap.

Request body

Field	Type	Required
`question_type`	string	yes
`answer_value`	string	
`rating_value`	string	
`survey_id`	string	
`question_id`	string	
`page_url`	string	

Responses

Code	Description
`200`	Success.

400	Bad request — `bad_request`.
401	Unauthenticated — `invalid_key`, `revoked_key`.
402	Payment required — `account_inactive`.
403	Forbidden — `origin_forbidden`.
413	Payload too large — `payload_too_large`.
429	Rate limited — `rate_limited`.

`GET /v1/surveys`

Survey results

The dashboard READ of structured survey answers: per-question aggregates (choice bars + rating averages) plus recent open-ended survey comments. Dashboard-authed like `/v1/usage`.

Responses

Code	Description
200	Success.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `forbidden_key`.

`GET /v1/tags`

List custom tags

The account's custom tag vocabulary (`tags`) and the per-account cap (`max`).

Responses

Code	Description
200	Success.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `forbidden_key`.

`POST /v1/tags`

Create a custom tag

Add a custom tag to the account vocabulary. Owner/admin only; 409 on a duplicate name or when the per-account cap is reached. **Owner only.**

Request body

Field	Type	Required
`name`	string	yes
`description`	string	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`409`	Conflict — `duplicate_tag`, `tag_cap`.

`POST /v1/tags/{id}`

Edit or delete a tag

Edit a tag's description and/or soft-delete it (`deleted: true` hides it from the vocabulary without dropping the labels already applied to conversations). Owner/admin only; at least one of `description`/`deleted` must be supplied, and `404` if no tag matches the id. **Owner only.**

Parameters

Name	In	Type	Required	Description
`id`	path	string	yes	Tag id — the trailing path segment after `/v1/tags/` (≤100 chars).

Request body

Field	Type	Required
`description`	string	
`deleted`	boolean	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.

`GET /v1/team`

Members & invites

Returns the account’s members. Owners/admins also see pending invites; plain members see members only.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/team/invite`

Invite a teammate

Owner/admin only. Emails a magic invite link (valid ~7 days). Owners may invite admins; admins may invite members only. Invites never grant owner. Returns 409 without sending when the plan’s editor seats are already in use; acceptance rechecks the snapshotted allowance atomically. Otherwise the response is a neutral 200 regardless of delivery.

Request body

Field	Type	Required
`email`	string	yes
`role`	string (member | admin)	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`, `seat_limit`.
`429`	Rate limited — `rate_limited`.
`503`	Service unavailable — `preview_disabled`.

`POST /v1/team/invite/revoke`

Cancel an invite

Owner/admin only. Cancels an unconsumed invite for an email. Idempotent (revoked: 0 when none pending).

Request body

Field	Type	Required
`email`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.

`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.

`POST /v1/team/remove`

Remove a member

Owner/admin only, with a last-owner guard. Admins may remove non-owners. Removal revokes the member's keys and deletes their invites first.

Request body

Field	Type	Required
`user_id`	string	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `last_owner`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.

`POST /v1/team/role`

Change member role

Owner/admin only, with a last-owner guard. Non-owners cannot grant or alter owner/admin. Demotions revoke the member's keys before the change.

Request body

Field	Type	Required
`user_id`	string	yes
`role`	string (owner | admin | member)	yes

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`, `last_owner`.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `csrf`, `forbidden`, `forbidden_key`.
`404`	Not found — `not_found`.

`GET /v1/threads`

List answer turns

A flat, cursor-paged feed of answer turns in ascending `(ts, conversation_id, turn)` order (oldest first) — built for forward incremental sync. Page with `next_cursor` / `has_more`.

Parameters

Name	In	Type	Required	Description
`since`	query	integer		Cursor: return turns at or after this epoch-ms timestamp (inclusive lower bound), or pass a prior page's `next_cursor`.
`limit`	query	integer		Max turns to return.

Responses

Code	Description
`200`	Success.
`401`	Unauthenticated — `invalid_key`.
`403`	Forbidden — `forbidden_key`.

`POST /v1/transcript`

Record a Q&A turn

Upserts one conversation turn (question + answer, optionally the model's reasoning) keyed on (key, conversation_id, turn). At least one of question/answer is required. Best-effort over the per-account daily cap; an out-of-range `turn` is rejected rather than coerced.

Request body

Field	Type	Required
`conversation_id`	string	yes
`turn`	integer	
`question`	string	
`answer`	string	
`reasoning`	string	
`sources`	object	

Responses

Code	Description
`200`	Success.
`400`	Bad request — `bad_request`.
`401`	Unauthenticated — `invalid_key`, `revoked_key`.
`402`	Payment required — `account_inactive`.
`403`	Forbidden — `origin_forbidden`.
`413`	Payload too large — `payload_too_large`.
`429`	Rate limited — `rate_limited`.

`GET /v1/transcripts`

List transcripts

Returns the account's stored transcripts grouped into conversations (newest first), each turn carrying its vote. Capped at ~300 turns; `truncated` signals older history exists.

Responses

Code	Description
200	Success.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `forbidden_key`.

`GET /v1/usage`

Account snapshot

The dashboard overview payload: account status, balance, recent usage rows, top-ups, reader feedback/comments, account events, and the payment/auto-top-up configuration. Readable even when the account is suspended.

Responses

Code	Description
200	Success.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `forbidden_key`.

`GET /v1/usage/members`

Billing usage by member and account key

Owner/admin billing usage grouped by the team member who minted each key, over an optional [since, until) window. Account-owned secret/admin/CLI usage is listed as one row per key with safe key metadata.

Parameters

Name	In	Type	Required	Description
`since`	query	integer		Inclusive lower bound, epoch ms (default: 30 days ago).
`until`	query	integer		Exclusive upper bound, epoch ms (default: now).

Responses

Code	Description

200	Success.
400	Bad request — `bad_request`.
401	Unauthenticated — `invalid_key`.
403	Forbidden — `forbidden`, `forbidden_key`.

Support

Top questions

How do seats work?

Seats cover editors and admins, readers are always free and unlimited, and Pro and Business can buy add-on seats when they need more.

What happens when I cancel?

Your plan runs to the end of the paid period, then the account returns to Free with your prepaid balance untouched.

What happens when I use up my included AI for the month?

By default paid AI pauses and readers fall back to free models, so you're never charged a surprise overage.

What if I don't use all of it — does it roll over?

Unused allowance rolls over up to one month's worth, so allowance plus rollover never exceeds about two months at once.

What's the most I could be billed in a month?

Your maximum includes the plan fee, fixed add-on-seat charges, and any overflow already incurred; pay-as-you-go also includes your remaining cap headroom.

All articles

Can I remove my saved card while I'm on a subscription?

Yes — removing your card also detaches it from your subscription's billing.

Can I still self-host on Pro or Business?

Yes — managed hosting is included on paid plans, never required.

Do I need my own model API keys?

No — all metered AI runs through Aardvark's managed keys on every plan.

How do I add or remove seats?

Add or remove seats from Billing → Seats, with prorated charges and credits on your next invoice.

How do seats work?

Seats cover editors and admins, readers are always free and unlimited, and Pro and Business can buy add-on seats when they need more.

How does annual billing save money?

The annual discount applies to the platform fee, while the included-AI allowance is funded at full value every month.

How does Enterprise billing work?

Subscribe self-serve at the published price, or arrange a signed contract with negotiated terms billed out of band.

Is the cutoff an exact-to-the-penny $0 guarantee?

The cutoff stops within roughly one request's cost of your limit, so keep a small buffer if you need a hard ceiling.

What are my options when the included AI runs out?

Choose cap-and-hold, pay-as-you-go with a cap you set, or a full shutoff — switchable anytime on the Billing page.

What exactly counts against the included AI?

The billed dollar cost of metered requests counts against your allowance, pooled account-wide and drained before your prepaid balance.

What gets used first — my allowance or my prepaid balance?

Rolled-over allowance drains first, then this month's allowance, and your prepaid balance only if you've opted into overflow.

What happens to my add-on seats when I change plans?

Add-on seats come with you — re-priced immediately on upgrades and at your next renewal on downgrades.

What happens to my prepaid balance if I cancel, downgrade, or a payment fails?

Your prepaid balance is always yours — cancelling, downgrading, or a lapsed payment never spends it.

What happens when I cancel?

Your plan runs to the end of the paid period, then the account returns to Free with your prepaid balance untouched.

What happens when I run out of seats?

Invites are simply blocked until you buy add-on seats or remove a member — your bill never grows on its own.

What happens when I use up my included AI for the month?

By default paid AI pauses and readers fall back to free models, so you're never charged a surprise overage.

What if a subscription payment fails?

You get about two weeks of automatic retries before the plan lapses to Free, and your balance is never used to cover it.

What if I don't use all of it — does it roll over?

Unused allowance rolls over up to one month's worth, so allowance plus rollover never exceeds about two months at once.

What if I fire a lot of AI requests at the same time?

Concurrent paid requests are capped per account; extras get a brief 429 with a Retry-After and can be retried once a slot frees.

What if I schedule a downgrade and then change my mind?

Switch back before your renewal and you stay on your current plan at your original price.

What if I want to raise the cap again later — do I re-confirm every time?

Once acknowledged, later cap changes never prompt you again.

What if our published prices change after I subscribe — does my bill go up?

You keep the price you signed up at for as long as you stay on that plan.

What is auto top-up?

An optional card-on-file feature that refills your prepaid balance automatically when it runs low.

What's the most I could be billed in a month?

Your maximum includes the plan fee, fixed add-on-seat charges, and any overflow already incurred; pay-as-you-go also includes your remaining cap headroom.

When do plan changes take effect?

Upgrades apply immediately with proration; downgrades and cancellations wait for your next billing date.

Why did my AI pause with an “unusual usage” message?

An automatic safety brake pauses paid AI when spending spikes far above your normal pattern — one click resumes it.

Why do I have to confirm a checkbox to raise my overflow cap past 2× my included AI?

The one-time checkbox is a guardrail so a cap far above your allowance is a deliberate choice, not a silent overage.

Anatomy of a Mantine island

Every interactive widget on this site — the cards, the accordions, the changelog timeline — starts life as a tag in a Markdown file. This post follows one of them from source to screen.

An island begins as either a tag like `{% card %}` or a `component('aardvark', 'card', …)` call in a `{% %}` block — one implementation behind both. At build time, the tag doesn't render HTML directly; it emits a **placeholder**: a wrapper element carrying a `data-aardvark-island` marker naming the component, with the props serialized alongside. That marker is also your styling hook — CSS like `[data-aardvark-island="Card"]` targets every rendered card on the site.

The client side is a single islands bundle, built with esbuild, that scans the page for those markers on load and mounts the matching React component — real Mantine components, with the theme palette seeded from your site's SCSS colors.

The interesting part is what happens between build and load. With `islands.ssr: true` (this site has it on), the build prerenders each island to **static HTML** using Node, esbuild, and linkedom — so the actual widget markup is baked into the `.html` files. That matters for three audiences: crawlers see real content instead of an empty div, no-JS readers get a usable page, and everyone else gets a correct first paint instead of a layout shift. When the client bundle loads, it re-renders the same component over that markup, and the island becomes interactive.

The result is a static site that ships real HTML and also behaves like a React app where it counts. Browse the component library to see every island available, or read the changelog entry that introduced them.

Designing the honest AI meter

Most products meter AI in **credits** — a currency you can't reason about, converted at a rate you can't see. When we designed Aardvark's AI billing, we set one constraint up front: every number a customer sees is a **dollar**. Here's what that constraint produced.

Published prices, real arithmetic. The pricing table lists what every model costs per answer on every plan — real figures, no conversion math. Paid plans include a monthly allowance denominated in those same billed dollars, and the dashboard shows dollars included, dollars used, your own trailing burn rate, and an estimated depletion date. No "≈N answers" marketing math.

The worst case is written down. The default when your included AI runs out is **cap-and-hold**: paid AI pauses, readers fall back to free models, and your max possible bill is your subscription price plus any fixed add-on-seat charges and any overflow already incurred this month. If you opt into pay-as-you-go overflow, the dashboard also includes whatever remains under the cap you set. Raising that cap past 2× your allowance asks for an explicit acknowledgment, once, because a large overage should be a decision rather than a surprise.

Guardrails assume something will go wrong. An automatic safety brake pauses paid AI when spending runs far above your account's normal pattern — one click resumes it. Concurrent paid requests are capped per account. And we're precise about our own precision: the cutoff stops within roughly one request's cost of your limit, not to the exact cent, and the docs say so.

Some smaller choices fell out of the same principle. Unused allowance rolls over one month, capped at about two months banked, so a quiet period can't build into a bill-shaped balloon. Answers served by `:free` models never bill at all. And a paid plan's metered AI never touches your prepaid balance unless you opt into overflow — on Free, where AI is pay-as-you-go, each metered answer draws that balance directly.

None of this is complicated. It's mostly the discipline of refusing a layer of abstraction that only ever benefits the seller. The full details — every edge case, in plain language — live in the support knowledge base.

Introducing Ask AI, the reader assistant

With 0.8.0, every Aardvark site can ship a native **Ask AI** assistant. A floating panel sits on every page; readers ask a question in natural language, and the assistant answers from **your content**, citing the pages it used. Each answer can be rated / , and readers can attach images, PDFs, or code when a screenshot says it better than a sentence.

There is no third-party widget to embed. The assistant is a first-party feature that calls the Aardvark cloud gateway, which proxies the model, meters the spend, and records the conversation for analysis. Enabling it is two lines of config:

```
ai:
  assistant:
    enabled: true
```

plus a spend-capped public gateway key baked in as a build-time environment variable — never a provider key in your repo.

Two design choices matter most. First, **grounding**: when your corpus fits the model's context budget, the assistant inlines the whole thing on the first turn and answers with zero fetches; when it doesn't, it navigates your docs page-by-page, reasoning about which page to read next. Either way the answer comes from what you actually published. Second, **honest metering**: usage bills in real dollars against a capped allowance, so a public docs site can offer AI answers without signing up for an unbounded bill. Bring your own key, or try it on the bundled, capped allowance.

Behind the panel sits a conversation-analytics dashboard — top questions, coverage gaps, intent tags — so every reader question becomes product insight. You're reading the docs of a site that runs it: press `Cmd+I` and ask something.

Read more on the AI assistant page, or see the changelog entry for the release details.

Multi-language docs in practice

Since 0.6.0, Aardvark serves translated sites from per-language directories — and this site is a live example: the English content lives in `content/`, the French mirror in `content-fr/`, served under `/fr` with an automatic language picker in the header. Here's the setup, start to finish.

First, declare your languages in `aardvark.config.yaml`:

```
languages:
  base:
    code: en
    label: English
  others:
    - code: fr
      label: Français
      source: content-fr
```

That's the whole routing story. The base language builds at the site root; each additional language builds under its code prefix, and the picker appears on every page.

The harder problem is keeping translations **filled and fresh**, and that's where `vark build --translate` comes in. It compares the translated tree against the base content and fills in pages that are missing or changed — not the whole site, every time — using a model, with the results written into your language directory as ordinary Markdown you can review and edit. A `--retranslate-all` flag exists for when you want a clean sweep.

Two things keep the output trustworthy. Unchanged pages are skipped via a **content-hash cache**, so a page translated once isn't re-translated until its source (or the glossary) changes. And the site's definitions glossary feeds the translator, so your product terms — the words you don't want localized creatively — survive intact.

Treat the translated tree like any other content: it's plain Markdown in your repo, it diffs in review, and hand edits are kept for any page whose source hasn't changed — a page whose source does change is re-translated wholesale. One caveat: the "unchanged" bookkeeping lives in the local build cache, so on a fresh machine (or after clearing the cache) a `--translate` run re-translates everything — commit your edits and review that run's diff rather than letting it land blind. Details are in the CLI reference and the changelog entry.

vark 0.9 release roundup

vark 0.9.0 landed this week. Here's what's in it, plus the highlights from the 0.8 line that led up to it — the full timeline is always on the changelog.

The CLI reference now writes itself. The CLI reference page is generated straight from `vark --help`: every flag and subcommand is documented from its own `--help` output, so the docs stay in lockstep with the code. No hand-maintained option tables, no drift — if a flag ships, its documentation ships with it. It's the docs-tooling equivalent of eating your own dog food, and it's how this site's own CLI page is built.

The edit loop got faster. `vark dev` keeps rebuilds tight by skipping the build's heaviest phases — Open Graph card rendering and the whole-site PDF — which only matter when you publish. (Full per-page incremental skipping is a promise we're deliberately not making yet; we wrote up why in Why Aardvark rebuilds every page (for now).)

And from the 0.8 series, in case you missed it:

- **Standardized code blocks** — every code block on the page, in the file-tree modal, and in OpenAPI samples now shares one **Copy** and **Download** affordance.
- **OpenAPI search indexing** (0.8.2) — operation and schema descriptions from your spec are part of the search index, so readers searching for an endpoint's behavior land on the right reference section.
- **Ask AI** (0.8.0) — the optional reader assistant that answers questions grounded in your docs. We introduced it properly in its launch post.

Upgrade with your package manager of choice, and if anything regresses, the changelog is the quickest way to see what moved.

Why Aardvark rebuilds every page (for now)

Here's an honest engineering answer to a question we get: does `vark` do incremental builds? Today, no — every build renders every page. The edit loop stays fast a different way: `vark dev` skips the build's heaviest phases entirely (Open Graph card rendering and the whole-site PDF, which only matter when you publish), and the render pipeline itself is lean enough that a full re-render of a mid-sized site lands in seconds.

Why not skip unchanged pages? Because on a docs site, "unchanged" means much more than "the Markdown file has the same bytes."

A rendered Aardvark page is a function of many inputs. The obvious one is the page's own source — prose, front matter, the `` tags inside it. But pages also read **shared inputs**: the site config (a renamed tab changes every page's header), the theme (one SCSS variable recolors the whole site), data files loaded through templating, snippets and includes, and the navigation itself — adding one page inserts a sidebar entry on all of its siblings. Skip a page whose own file is unchanged while one of those shared inputs moved, and you ship a stale build that looks fine until someone notices the sidebar is missing an entry.

So any future page cache has to be conservative: a page can only be skipped when everything that could affect its output is accounted for, and anything site-wide must invalidate broadly rather than cleverly. That's a deliberate trade. A cache that is occasionally too eager to rebuild costs you seconds; a cache that is ever too eager to skip costs you correctness, and a docs generator that sometimes publishes stale pages is worse than a slow one. Until we can make the skip provably safe, rebuilding everything — and keeping "everything" cheap — is the honest default.

The dev-loop details live in the `dev` command docs, and the changelog has the release notes.

Changelog

Everything new in Aardvark, newest first. Releases carry a version badge; use the tag cloud on the right to filter the timeline by area — pick more than one to combine them.

Generate the CLI reference from `vark --help` — May 28, 2026, 4:30 PM

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Faster dev-loop rebuilds — May 28, 2026, 9:15 AM

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Standardized code-block Copy & Download buttons — May 22, 2026

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

Index OpenAPI descriptions for search — May 14, 2026

Operation and schema descriptions from your OpenAPI spec are now part of the search index, so a reader searching for an endpoint's behavior lands on the right reference section — not just pages that happen to mention it in prose.

Ask-AI reader assistant — May 3, 2026

An optional **Ask AI** assistant answers reader questions grounded in your docs, backed by a metered cloud gateway. Bring your own key, or use the bundled, capped allowance to try it out.

Build-time accessibility contrast audit — April 19, 2026

`vark build` now runs a non-fatal **WCAG color-contrast audit** over your theme colors (light and dark) and reports any pair that falls below the configured level, so regressions surface before they ship. Read more under Accessibility.

Social unfurl cards (Open Graph) — April 5, 2026

Every page can render a branded **1200×630 Open Graph card** — your logo over a gradient of the theme's primary color — rendered fresh under `_aardvark/og/` on every production build, and skipped entirely by `vark dev` so the editing loop stays fast.

Multi-language sites — March 21, 2026

Serve translated content from per-language directories with an automatic language picker. `vark build --translate` fills missing or changed pages with a model, grounded in the site's definitions glossary; unchanged pages are skipped via a content-hash cache.

Interactive Mantine islands — March 2, 2026

Embed any **Mantine** component as an interactive island straight from Markdown. Islands are optionally prerendered to static HTML at build time for crawlers and first paint, then rehydrated on the client for full interactivity.

Blog

Designing the honest AI meter

Why Aardvark bills AI in dollars instead of credits, and the guardrails that keep a metered feature from ever producing a surprise bill.

Anatomy of a Mantine island

How a Markdown tag becomes a prerendered, rehydrated React component — the full journey from source to interactive widget.

Why Aardvark rebuilds every page (for now)

The edit loop stays fast by skipping the heavy publish phases — and per-page incremental skipping is a harder promise than it looks.

vark 0.9 release roundup

The 0.9 release in one read — a self-generating CLI reference, a faster dev loop, and the best of the 0.8 line.

Introducing Ask AI, the reader assistant

Aardvark 0.8 ships a native Ask AI panel that answers reader questions from your own docs, with cited sources and metered, dollar-based billing.

Multi-language docs in practice

A working setup for translated docs — per-language content directories, an automatic language picker, and model-filled translations grounded in a glossary.

Build-time accessibility contrast audit

`vark build` now runs a non-fatal **WCAG color-contrast audit** over your theme colors (light and dark) and reports any pair that falls below the configured level, so regressions surface before they ship. Read more under Accessibility.

Ask-AI reader assistant

An optional **Ask AI** assistant answers reader questions grounded in your docs, backed by a metered cloud gateway. Bring your own key, or use the bundled, capped allowance to try it out.

Generate the CLI reference from `vark --help`

The **CLI reference** page is now generated straight from `vark --help`, so every flag and subcommand stays in lockstep with the code:

- No hand-maintained option tables to drift
- Each subcommand is documented from its own `--help`

See the CLI reference for the full command list.

Standardized code-block Copy & Download buttons

Every code block — on the page, in the file-tree modal, and in OpenAPI request/response samples — now shares one **Copy** and **Download** affordance, so the interaction is identical everywhere a reader meets code. See the components library.

Faster dev-loop rebuilds

`vark dev` keeps edit-loop rebuilds fast by skipping the build's heaviest publish phases — Open Graph card rendering and the whole-site PDF — which only matter when you ship. Full `vark build` output is unchanged. Why Aardvark still re-renders every page covers the design thinking.

Interactive Mantine islands

Embed any **Mantine** component as an interactive island straight from Markdown. Islands are optionally prerendered to static HTML at build time for crawlers and first paint, then rehydrated on the client for full interactivity.

Multi-language sites

Serve translated content from per-language directories with an automatic language picker. `vark build --translate` fills missing or changed pages with a model, grounded in the site's definitions glossary; unchanged pages are skipped via a content-hash cache.

Index OpenAPI descriptions for search

Operation and schema descriptions from your OpenAPI spec are now part of the search index, so a reader searching for an endpoint's behavior lands on the right reference section — not just pages that happen to mention it in prose.

Social unfurl cards (Open Graph)

Every page can render a branded **1200×630 Open Graph card** — your logo over a gradient of the theme's primary color — rendered fresh under `_aardvark/og/` on every production build, and skipped entirely by `vark dev` so the editing loop stays fast.

Built-in Components

Aardvark ships a built-in `{% tag %}` for **every Mantine core component** — no setup, no JavaScript to write. They're grouped by category in the left nav, mirroring Mantine's own groupings, so you can browse by what a component does. On top of the Mantine set, Aardvark adds a handful of **native** components built just for documentation sites, plus a growing **Community Components** catalog: eligible third-party Mantine extensions are wrapped as built-in tags, while notable policy exclusions are documented explicitly.

Pick a category to see every tag in it, with live examples and source:

✨ Aardvark Extras

Native components with no Mantine equivalent — banners, gitfolders, includes, maps, and changelogs.

Layout

Structure a page — containers, stacks, grids, spacing, surfaces, and scroll areas.

Inputs

Text fields and controls — inputs, selects, checkboxes, switches, sliders, and color pickers.

Combobox

Select, autocomplete, multi-select, and tag inputs built on Mantine's Combobox.

Buttons

Buttons and button-like controls for actions and links.

Navigation

Move through content — tabs, steps, breadcrumbs, pagination, and tree views.

Feedback

Tell the reader what happened — callouts, progress, loaders, and notifications.

Overlays

Modals, drawers, dialogs, tooltips, popovers, and menus layered over the page.

Data Display

Show content — cards, badges, images, tables, accordions, and timelines.

Typography

Style text — titles, code, highlights, lists, dividers, and tables.

Community Components

Third-party Mantine extensions — bundled tags that pass the license and compatibility gates, plus documented exclusions.

Build your own

When the built-ins don't cover it, define your **own** component in your project — that's a custom component, and it's where the real flexibility is. Drop a React component in `snippets/` and call it from Markdown exactly like a built-in:

`snippets/ProductCard.jsx` freely blends **plain HTML** (`<div>`, `<h3>`, `<p>`) with Mantine components (`Card`, `Group`, `Badge`, `Button`) — inside a snippet it's just React, so any blend of HTML and Mantine you choose works. See Components & snippets for the full authoring guide.

Need a whole **third-party React library** — Stripe Elements, a charting kit, an icon set? A theme can pull one in and address it through the same tag with a library-name first argument: `{% component('stripe', 'PaymentElement') %}`. See Component libraries.

How do I add or remove seats?

From **Billing → Seats**. Adding seats raises your limit **immediately** and prorates the cost onto your next invoice, so you only pay for the part of the period you actually use them. Removing seats posts a proration **credit** to your next invoice — first drop the members or pending invites you no longer need on the Team tab (we won't let you reduce below the seats currently in use), then lower the count. Everyone already on your team keeps their access either way.

What happens to my add-on seats when I change plans?

They come with you. On an **upgrade** your add-on seats move to the new plan’s per-seat rate right away (prorated onto your next invoice, like the plan change itself); on a **downgrade** they switch over at your **next renewal**. If you move to a plan that doesn’t sell add-on seats, the add-ons are removed at that point and you stop being billed for them — the people over the new limit keep their access, you just can’t invite past it until you’re back under.

What gets used first — my allowance or my prepaid balance?

In order: any **rolled-over** allowance first (it's the oldest and expires soonest), then **this month's** allowance, and only then — if you've turned on pay-as-you-go overflow — your **prepaid balance**. Your balance is never touched unless you opt in.

How does annual billing save money?

The discount applies to the platform fee; your included-AI allowance is funded at full value every month either way (it’s real usage, not a discountable line item). The allowance still resets monthly on annual plans.

What is auto top-up?

Optional. With a card on file, we automatically refill your prepaid balance when it runs low — so free-tier and pay-as-you-go usage aren’t interrupted. You choose the amount and can turn it off anytime.

What if I fire a lot of AI requests at the same time?

We cap how many paid requests run at once per account to keep spending predictable. Extra concurrent requests get a brief `429` with a `Retry-After` header — retry after it, possibly more than once while the account is still at its cap; free-model requests aren't limited this way.

How does Enterprise billing work?

Enterprise works two ways. You can **subscribe to it self-serve** at the published price, just like Pro or Business, with the saved card on your dashboard. Or, for a **signed contract** with a negotiated allowance, seats, and member rate, we can provision it out of band — billed separately, with no card on file for the plan, and your dashboard reflecting your contract terms. Either way your “max possible bill” is shown for the plan you’re on.

What if a subscription payment fails?

You get a grace period of about two weeks with automatic retries before the plan lapses to Free — and, again, your prepaid balance is never used to cover it.

How do seats work?

Seats cover **editors and admins** — the people who create and manage your docs. **Readers are always free and unlimited** on every plan. Each plan includes a set number of seats (Free 1, Pro 5, Business 12, Enterprise unlimited), and on Pro and Business you can buy **add-on seats** when you need more: **$10/seat/mo on Pro**, **$15/seat/mo on Business** (annual plans bill seats at 12× the monthly rate, so the yearly total matches the monthly one).

What if I don’t use all of it — does it roll over?

Yes, up to one month’s worth. Anything beyond that doesn’t keep stacking, so your allowance plus rollover never exceeds about two months at once — a deliberate cap so a quiet period can’t build into an unexpectedly large bank.

What’s the most I could be billed in a month?

With **cap-and-hold**, the figure is your plan fee plus any fixed add-on-seat charges and any overflow already incurred this month. With **pay-as-you-go**, it also includes whatever remains under the overflow cap you set. Your dashboard shows this “max possible bill” figure directly. (Usage-based build/compute, if you use it, draws your prepaid balance separately and isn’t part of that number.)

Why do I have to confirm a checkbox to raise my overflow cap past 2× my included AI?

It’s a guardrail against a surprise bill. Setting a cap far above your included allowance is a deliberate choice, so we ask you to acknowledge it once rather than let a large overage happen silently.

What are my options when the included AI runs out?

Three, and you can switch anytime on the Billing page:

- **Cap-and-hold** (default) — paid AI pauses; readers keep getting free-model answers. Your maximum is your plan fee plus any fixed add-on-seat charges and any overflow already incurred this month.
- **Pay-as-you-go** — paid AI keeps running past your allowance, drawing your prepaid balance at your plan's member rate, up to a monthly cap you set.
- **Shutoff** — all AI stops (including free models) until next month's allowance arrives.

Do I need my own model API keys?

No — on every plan, all metered AI runs through Aardvark's managed keys. You never create, rotate, or secure a provider key, and there is no bring-your-own-key tier to manage.

What happens to my prepaid balance if I cancel, downgrade, or a payment fails?

It’s always yours. Cancelling, downgrading, or a lapsed subscription payment never spends or absorbs your prepaid balance — it’s kept completely separate from your plan.

What if our published prices change after I subscribe — does my bill go up?

No. You keep the price you signed up at for as long as you stay on that plan. If we later change our published pricing, your live subscription isn’t moved — the price you agreed to is the price you pay.

What if I want to raise the cap again later — do I re-confirm every time?

No. Once you've acknowledged it, it stays acknowledged; later cap changes don't prompt you again.

Can I remove my saved card while I’m on a subscription?

Yes. Removing it also detaches it from your subscription’s billing, so nothing keeps charging a card you’ve taken off.

What happens when I run out of seats?

Your bill never grows on its own — it's **block-and-buy**. When every seat is taken, new invites are simply **blocked** until you make room, and nothing is charged and no one loses access until you decide to add seats. To fit another teammate you either buy add-on seats or remove a member or pending invite on the **Team** tab. (On an operator-provisioned or contract plan, seats are arranged through support rather than self-serve.)

Can I still self-host on Pro or Business?

Yes. Managed hosting is included, never required.

Is the cutoff an exact-to-the-penny $0 guarantee?

It stops within roughly one request’s cost of your limit, not to the exact cent — the final request is admitted while you’re still in the black and then billed. If you need a hard ceiling, keep a small buffer.

What if I schedule a downgrade and then change my mind?

Just switch back to your current plan before your next renewal. You stay exactly where you were, at your **original price** — even if our catalog prices happened to change in the meantime. Undoing a scheduled downgrade never quietly re-prices you.

Why did my AI pause with an “unusual usage” message?

An automatic safety brake noticed spending far above your account’s normal pattern and paused paid AI to protect you from a runaway bill. It isn’t a funds problem — one click on the Billing page resumes it.

What happens when I use up my included AI for the month?

By default, paid AI **pauses** for the rest of the month and readers automatically fall back to free models — so you're never charged a surprise overage. It resets on your next monthly date. If you'd rather keep paid AI running past your allowance, turn on pay-as-you-go overflow.

What exactly counts against the included AI?

The billed cost of metered requests — the same number the dashboard meter and your usage ledger show, measured in real dollars, not opaque credits. It pools across your whole account and drains before your prepaid balance.

What happens when I cancel?

Your plan stays active until the end of the period you've already paid for, then the account returns to Free pay-as-you-go. You can **resume anytime before then** and nothing changes. Your prepaid balance is untouched. (An operator-provisioned Enterprise/comped plan ends immediately instead, with no scheduled end date.)

When do plan changes take effect?

Upgrades apply right away, with the difference prorated onto your next invoice. Downgrades and cancellations take effect at your **next billing date** — you keep everything you paid for until then, and nothing is prorated. Unused prepaid balance is always yours either way.