CompMD Specification v1 (draft)

CompMD (Component Markdown) is a Markdown flavor that adds components, containers, slots and CSS-like layout to CommonMark + GFM, while staying readable in any plain Markdown renderer.

Version 1, draft. Last changed 2026-11-03. Every change is listed in the changelog.

Author: Hakan Serce.

Website: compmd.dev, with this spec as a page and a playground to try CompMD in.

---

1. Status and conventions

This is a working draft of CompMD version 1. It specifies document format only; parser and renderer implementations are out of scope.

- The key words **MUST**, **MUST NOT**, **SHOULD** and **MAY** are used as in RFC 2119.
- Terms are defined where they are introduced, and listed in section 1.1.
- Grammar uses ABNF-style notation: `SP` is one space, `EOL` is end of line, `"x"` is a literal, `[ ]` is optional, `*` is zero or more.
- "Line" means a source line after CommonMark tab expansion; a component line **MAY** be indented up to 3 spaces, as CommonMark fences are.

1.1 Terms

- **Binding.** The host contract mapped to one framework: plain JavaScript, React and Svelte 5 in version 1 (section 7.6).
- **Box.** The rectangle a block is laid out in. Every component invocation that is a block has one *outer box* (section 9.2): a host-language component's `target`, or the box holding a `compmd` component's body. A `display=contents` component has none.
- **Breakpoint.** One of the widths `sm` (480 px), `md` (768 px) and `lg` (1024 px). A param with a breakpoint prefix, such as `md:gap=l`, applies from that width up, measured on the block's nearest box (section 9.3).
- **Built-in.** A component defined by this spec, in the `@compmd/core` pack (section 9).
- **Code renderer.** A leaf whose signature also names a language with `renders`, so it renders the code blocks of that language, the block's text as its `code` param (section 7.7).
- **Component.** A named, reusable piece of content invoked from a document: a container, a leaf or an inline component (section 4), defined in CompMD or in a host language

(section 7).

- **CompMD processor.** Any tool that parses CompMD: a renderer, a linter, a formatter. Section 13 defines what conforming parsers and processors implement (section 13).
- **Container.** A component written as `::: name` … `::: /name`, with a body (section 4.1). The box a block's breakpoints follow is its *nearest box*, not a container.
- **Expanded tree.** The result of stage 4b: the document's content, with definitions, `if`, `each`, `value` and interpolations replaced by what they produce (section 12.3).
- **Fallback.** What shows in place of a component that can't be rendered: a leaf's or inline component's text in `[…]`, a container's slots as plain Markdown (section 4.5).
- **Host contract.** How a processor that renders HTML runs a host-language component, whatever its framework: `mount`, `update` and `destroy` (section 7.4).
- **Host language.** A programming language a component can be defined in, other than CompMD: `js`, `ts`, `jsx`, `tsx` and `svelte` in version 1 (section 7).
- **Input.** A typed value the reader can change, declared in the frontmatter's `inputs` key under `md`, and read as `md.input.<name>` (section 8.7).
- **Item.** One top-level block in a layout container's default slot (section 9.1) or in a host-language component's slot (section 7.4).
- **Nearest box** (of a block). The closest box around it in the expanded tree (section 12.3): a `box` (a `compmd` component's box among them), a host-language component's outer box, or one item of a host-language component's slot. List items, quotes, table cells, `each`, `if`, the slots of `compmd` components and `display=contents` components don't count. At the top level, it is the area the processor renders into. A positioned block is placed in its nearest box (section 9.2), and its breakpoints follow its nearest box's width (section 9.3).
- **Origin.** The range of the processed document that made an expanded node, with its occurrence (section 12.3).
- **Pack.** A document or published package that exports components (section 6).
- **Parse tree.** The result of stage 1: the document as written, as mdast nodes and CompMD's own (section 12).
- **Permissive sandbox.** The trust level of a pack the processor's configuration gives more than the strict sandbox: the network to the origins listed for it, and `host` (section 15).
- **Plain renderer.** Any CommonMark or GFM renderer without CompMD support (section 14).
- **Sandbox.** Where a host-language component's code runs when it isn't trusted: it sees only its contract, and reaches nothing else unless its pack is permissive (section 15).
- **Slot.** A named part of a container's body, started by `:: #name`. Content before the first slot line is the *default slot* (section 4.2).
- **Strict sandbox.** The default trust level: a sandbox with no network, in which code sees only its contract (section 15).
- **Template slot.** A slot rendered once per element of an array param (section 7.5).

- **Trust level.** How far a pack's code is trusted, which the processor's configuration sets per pack and a document can never raise: a strict sandbox, a permissive sandbox, or trusted (section 15).
- **Trusted.** The trust level of the host's own code, built with its application: it runs outside a sandbox, and treats its params and slots as untrusted input (section 15).

---

2. Documents

A CompMD document is a CommonMark + GFM document, usually with a `.comp.md` extension and a frontmatter key `md` naming its version.

Property	Value
File extension	`.comp.md`
MIME type	`text/markdown; variant=compmd` (RFC 7763 media type; `compmd` to be registered in IANA's Markdown Variants registry, RFC 7764, when version 1 is published)
Base syntax	CommonMark 0.31, GFM as GitHub renders it, alerts and math (section 2.3)
Encoding	UTF-8

What makes a file CompMD. A file is a CompMD document when its frontmatter has an `md` key, or when whoever opens it says so: the `.comp.md` extension or the media type above. A CompMD document without a version is version 1, permanently. A `.md` file without an `md` key is CommonMark, and CompMD processing never applies to it; processors **MAY** offer to read it as CompMD. Documents that are shared or renamed **SHOULD** keep the `md` key, so they stay CompMD on their own.

2.1 Frontmatter

Frontmatter is a YAML block at the start of the document, delimited by `---` lines. Everything CompMD reads is under one key, `md`; every other key belongs to the author.

```yaml
---
md:
  version: 1
```

```
  import:
    - from: "@myorg/ui@1"
      names: [card, callout]
    - from: ./chart.jsx
      names: [chart]
  lang: en
title: Release notes
version: "1.4"
---
```

The author's `version` is `md.doc.version`; CompMD's is `md.version` inside the `md` key, and the two never mix.

Key under `md`	Value
`version`	`1`, or a string with a minor version (`"1.1"`); defaults to `1`
`import`	component imports (section 6.1)
`export`	component exports (section 6.2)
`data`	data imports (section 8.6)
`inputs`	values the reader can change (section 8.7)
`lang`	the document's language, a BCP 47 tag; defaults to `en` (`md.lang`)
`dir`	the document's text direction, `ltr`, `rtl` or `auto`; defaults to the direction of `lang`'s script (`rtl` for `ar`, `he`, `fa`, `ur`, …, from CLDR)
`tz`	the time zone of `md.today()`, an IANA name such as `Europe/Berlin` (`md.tz`)
`syntax`	opt-in syntax the document uses, such as `[wiki-links]` (section 2.3)

- Every key outside `md` is the author's, readable as `md.doc.<key>`. `md.doc` never holds CompMD's own keys.
- CompMD adds keys only under `md`, so a later version never collides with an author's key, or with keys other tools read from the same frontmatter (`layout`, `permalink`).

YAML. Frontmatter is YAML 1.2 with the core schema: `no` is a string and only `true` and `false` are booleans, and `2026-10-21` is a string. Tags (`!!…`), anchors and aliases (`&a`, `*a`) and duplicate keys are not allowed (F033). Processors **MAY** cap frontmatter size (F033). A leading byte order mark is ignored. Only `---` YAML is frontmatter; TOML (`+++`) is ordinary text. Write minor versions and other version-like values as strings: `1.10` is the number 1.1.

Metadata from a host. A processor **MAY** receive a document's metadata apart from its text, for example from a host that keeps it in its own fields. Such metadata is equivalent to the same keys in frontmatter. A document whose text has its own frontmatter uses only that; there is no merging, and a processor that was also given metadata reports W115. Since metadata has no place in the text, a diagnostic about it (an unknown `md` key, an import that can't be loaded) is reported at line 1.

2.2 Versions

- A processor **MUST** reject a major version it does not implement (F001).
- A minor version only adds to its major version. A processor reading a document with a later minor version than it implements reports W110 once and continues; features it doesn't know then fail with their own diagnostics.
- A key under `md` that a processor doesn't know is W111, and is ignored; the diagnostic names the nearest known key. Diagnostics that follow from the ignored key (an F014 for `md.input.x` after a misspelt `inputs`) name the W111.
- A processor **MAY** warn when a document uses a feature from a later minor version than it declares.

2.3 Base syntax

CompMD's base syntax is what GitHub renders, plus math. Each part is common in real documents and reads well in plain renderers.

Syntax	In compmd/1	Tree
CommonMark 0.31	always	mdast
GFM: tables, task lists, strikethrough, autolinks, footnotes (`[^1]`)	always	mdast
Alerts: `> [!NOTE]`, `[!TIP]`, `[!IMPORTANT]`, `[!WARNING]`, `[!CAUTION]`	always	a `blockquote` with `alert` set to the type in lower case (`"note"`), without the `[!NOTE]` line
Math: `$…$` inline, `$$…$$` and ```` ```math ```` fences for display	always	mdast `inlineMath` and `math`
Wiki links: `[[Page]]`, `[[Page|label]]`	opt-in, `md.syntax: [wiki-links]`	`wikiLink` with `target` and `label`

- **Math** follows Pandoc's rule (`tex_math_dollars`), in full:
  - An opening `$` is unescaped and has a non-space character on its right.
  - A closing `$` is unescaped, has a non-space character on its left and no digit after it, and is outside braces: in the math, an unescaped `{` opens a group and `}` closes one, and a `$` inside an open group doesn't close the math. A `}` with no group open is ordinary content.
  - Outside a group, an unescaped `$` with white space on its left (a space, a tab or a line break) ends the attempt: the opening `$` is text, and the next `$` may open math again. So a later valid closer never reaches back past a `$` that follows white space.
  - So "costs $5 and $10" is text; in "It costs $5 and $6. Math x^2", only `$x^2$` is math; `$x{y$ z}$` is the math `x{y$ z}`, and `$x{y $ z}$` the math `x{y $ z}`; and in `{{$doc.title}} by {{$doc.author}}` no `$` closes, since the second one is inside the `{{` that follows the first.
  - GitHub differs: it doesn't close math before a letter or digit, and it ignores braces, so `$x$y` is math here and text there. Authors who want the same result everywhere put a space or punctuation after a closing `$`.

  Every parser makes math nodes, so trees agree; a renderer without a math engine shows the TeX source. A fenced code block whose language is `math` is display math, as on GitHub.

- **Diagrams need no syntax.** A fenced code block keeps its language (`mermaid`, ...); a processor **MAY** render a code block by its language, as GitHub renders Mermaid. A document that imports a code renderer for a language renders its code blocks of that language through it (section 7.7); without one they stay code blocks.

- **Opt-in syntax** changes how existing text parses, so a document asks for it in `md.syntax`. `[[foo]]` next to a `[foo]: url` definition is a link inside brackets in CommonMark, and a wiki link only when the document opts in. What a wiki link's `target` points to is the processor's business; without pages to link to, it renders its label as text. In a GFM table, a wiki link's `|` is written `\|` (section 3.2). A name in `md.syntax` that a processor doesn't know is W111. Further opt-ins come in minor versions.

- **Links to a passage** of the document are ordinary links whose destination is a text fragment, the web's text directive (`#:~:text=[prefix-,]start[,end][,-suffix]`, each part percent-encoded): `[price: 12](#:~:text=price:%2012)`. A browser shows the passage on the rendered page. A tool that shows the document's source, such as an editor beside its preview, **MAY** select the passage in the source instead, its first match after the prefix and before the suffix, so a tutorial can point at the text to change. They are not CompMD syntax, and a plain renderer shows them as links.

- **Processors don't add syntax.** A CompMD document parses the same in every conforming parser. Anything else a processor offers is a component, for example in an

ambient pack (section 6.3): `:mark[text]` rather than `==text==`.

- **Raw HTML** is parsed as CommonMark says: HTML blocks and inline HTML are `html` nodes. A processor **MUST NOT** render raw HTML from a document its user doesn't trust; by default it sanitizes it with an allowlist like GitHub's (`details`, `summary`, `sub`, `sup`, `kbd`, `br`, `img` with `src` and `alt`, `a` with `href`; no scripts, styles or event attributes) or shows it as text. HTML is opaque to CompMD: no component lines inside an HTML block (section 3.2) and no interpolation in it (section 8.1).

---

3. Syntax

CompMD adds five block-level line forms and one inline form. The number of colons tells the form, and every form is recognizable from its own line, without lookahead.

Form	Marker	Example	Ends at
Container open	`:::`	`::: card {title="Setup"}`	its matching closer
Param continuation	`::: +`	`::: + icon=rocket`	end of line
Container close	`::: /`	`::: /card`	end of line
Leaf	`::`	`:: chart[Weekly installs] {src=installs.csv}`	end of line
Slot	`:: #`	`:: #footer`	end of line
Inline	`:`	`:badge[beta]{tone=info}`	the closing `]` or `}`

A code block's info string **MAY** also hold a param list after its language, ```` ```mermaid {theme=dark} ```` . It is read only when a code renderer renders the block (section 7.7); otherwise the info string stays as written, so other tools' conventions there (`{.class}`, `title="…"`) are untouched.

3.1 Grammar

```
component-line        = indent ( container-open / param-cont / container-close / leaf / slot ) *WSP EOL
```

```
indent          = 0*3SP                          ; counted from the content
column (section 3.2)
container-open  = ":::" 1*WSP name [ 1*WSP param-list ]
param-cont      = ":::" 1*WSP "+" 1*WSP param-body
container-close = ":::" 1*WSP "/" name
leaf            = "::" 1*WSP name [ fallback ] [ *WSP param-list ]
slot            = "::" 1*WSP "#" name
inline          = ":" name fallback [ param-list ]          ; no space
before "{"

fallback        = "[" inline-markdown "]"                  ; ends at its
matching "]" (section 3.2)
param-list      = "{" *WSP [ param-body ] *WSP "}"
param-body      = param *( 1*WSP param )
param           = key "=" value / "." name / "#" name / key
key             = [ name ":" ] name                        ; breakpoint
prefix, section 9.3
name            = ALPHA *( ALPHA / DIGIT / "-" / "_" )
value           = quoted / expr / json-array / json-object / bare
quoted          = DQUOTE *( string-char / escape / interpolation ) DQUOTE
escape          = "\" ASCII-punctuation
expr            = "(" cel-expression ")"                   ; CompMD CEL
profile, section 8.2
bare            = 1*bare-char
bare-char       = any Unicode character except white space, DQUOTE, "(",
")", "[", "]", "{" and "}"
```

Values. The first character of a value decides its kind, and each kind finds its own end:

First character	Kind	Ends at
`"`	string	the next `"` not escaped
`(`	expression	the matching `)`, counting parentheses outside CEL string and bytes literals (`"…"`, `'…'`, triple-quoted, `r"…"`, `b"…"`)
`[`, `{`	JSON array, object	the matching `]` or `}`, counting outside JSON strings; RFC 8259, on one line
anything else	bare	the next white space or `}`

- A param list ends at the first `}` that is not inside a value.

- In a string, a backslash escapes ASCII punctuation (`\"`, `\\`, `\{`); before any other character it is a literal backslash, so `"C:\dir"` is `C:\dir`.
- A bare value can't contain `(` or `{`: `grid-template-columns=repeat(3, 1fr)` and `title={{md.doc.title}}` are F011, whose message suggests quoting (`grid-template-columns="repeat(3, 1fr)"`, `title="{{md.doc.title}}"`) or an expression (`title=(md.doc.title)`).
- An empty value (`title=`) is F011; `title=""` is the empty string.
- After an inline component's `]`, `{{` starts an interpolation, not a param list.
- Names are ASCII and case-sensitive. Bare values and strings may hold any Unicode text.
- **Naming convention.** Component, slot and param names **SHOULD** be kebab-case (`habit-heatmap`, `side-panel`, `align-items`). `_` is allowed so names from elsewhere (file stems, data) need no renaming. Input and `as` names are always used in expressions, so they are CEL identifiers: one word or snake_case (`region`, `start_date`). F002's suggestion matches names regardless of case.
- **Names expressions can't use.** A param of a `compmd` definition (other than a `name` param, which binds a name and is never read, like a template slot's `as`), an `as` name and an input are variables in expressions, so they can't be CEL's reserved words (`as`, `break`, `const`, `continue`, `else`, `false`, `for`, `function`, `if`, `import`, `in`, `let`, `loop`, `namespace`, `null`, `package`, `return`, `true`, `var`, `void`, `while`) or its type names (`bool`, `bytes`, `double`, `duration`, `dyn`, `int`, `list`, `map`, `null_type`, `string`, `timestamp`, `type`, `uint`): F013. A param of a host-language component isn't an expression variable, so `chart {type=bar}` is fine.

3.2 Recognition rules

1. Component lines are recognized wherever a CommonMark fenced code block could start: at document level, inside block quotes, list items and CompMD containers. Indentation is counted from the content column of the enclosing list item or block quote; at 4 or more spaces past it, a line is indented code. Component lines interrupt paragraphs, so they are never lazy continuation lines.
2. A container opener or leaf directly after a paragraph line, without a blank line, is W108: plain renderers read it as part of the paragraph, which can change the structure around it (for example, join two lists that CompMD reads as separate). Closers, slot lines and continuation lines don't start new content, so they may follow a paragraph line directly.
3. Component lines and inline components are never recognized inside code spans, fenced or indented code blocks, HTML blocks, or link destinations. An HTML block runs to the next blank line, so in `<div>` followed directly by `::: card`, the `::: card` line is part of the HTML; a line inside an HTML block that starts with `::` is W116. A blank line after `<div>` ends the HTML block, and the card parses.

4. An inline component is recognized only when its `:` is at the start of a line or preceded by white space or by punctuation other than `:` and `\`. So `10:30[x]` and `std::vec[0]` are plain text.
5. An inline component's param list follows its `]` directly. `:badge[beta] {tone=info}` is a component without params followed by the text `{tone=info}`, and W107 reports it.
6. In a GFM table, a `|` inside an inline component or a wiki link is written `\|`, as GFM requires inside code spans, because GFM splits rows into cells first: `:badge[x]{show=(a \|\| b)}`, `[[Home\|start here]]`. Unescaped, the row gets an extra cell, which GFM drops when the table has fewer columns.
7. A line that begins (after its indentation) with `::` but matches no CompMD form, such as `:::card` or `:: /card`, is a **malformed component line**. Processors **MUST** report F016 for it rather than render it silently as text.
8. A backslash escapes CompMD syntax: `\::: card` and `\:badge[x]` are literal text, and the backslash is not rendered by CompMD processors.
9. A fallback ends at the `]` that matches its `[`: brackets nest, and `\[` and `\]` don't count. As in CommonMark's link text, a code span binds tighter than the brackets, so a `]` inside one doesn't end the fallback: `:a[b` c`]` `d]` has the fallback `b` `` `c]` `` ` d`, with the code span `c]`, as plain renderers show it. A run of backticks that no run of the same length closes is text, and its brackets count.

3.3 Canonical form (informative)

A formatter keeps a document's meaning, so it changes only whitespace and blank lines:

- one space after `:::` and `::`, and between params; one space before `{` on block lines, none on inline components: `:::card {title="A" tone=warn}` becomes `::: card {title="A" tone=warn}`;
- a blank line before and after block component lines, which also avoids W108.

It never changes quoting (a quoted value is always a string, a bare one is typed by the signature, so `count="3"` and `count=3` can differ), param order, how params are split across continuation lines, or the text.

4. Containers, leaves, inline components and slots

4.1 Containers

A container opens with `::: name` and **MUST** close with `::: /name`, where both names are identical. Its body is ordinary CompMD: paragraphs, lists, tables, code, leaves and further containers.

- Nesting is determined by names only; all fences use exactly three colons.
- A closer **MUST** match the innermost open container. Mismatch handling is in section 10.
- CompMD containers nest with CommonMark's containers: a CompMD container opens and closes inside the same list item or block quote, and lists, quotes and code nest freely inside it. A closer belongs to the list item or quote it is written in. A closer naming a container opened outside its list item or quote is F032; that container closes at the end of its own parent.

```
- Install

  ::: card {title="Setup"}
  Run `rayplay init`.
  ::: /card

> ::: callout {kind=tip}
> A callout in a quote.
> ::: /callout
```

- A container **MAY** be empty (opener directly followed by its closer).
- Containers nest to any depth. A container **MAY** contain another container of the same name.

4.2 Slots

A slot line `:: #name` inside a container starts the named slot; content runs until the next slot line or the container's closer.

- Content before the first slot line belongs to the **default slot**. No marker exists for it, so default content always comes first.
- A slot name **MUST** be declared by the component's signature (F006). Each named slot **MAY** appear at most once per container (F007).

- A slot line directly inside a container belongs to that container; a slot line outside any container is error F015.
- Slot order in the source does not set render order; the component decides where each slot renders.

4.3 Leaves

A leaf `:: name[fallback] {params}` is a block-level component with no body. It occupies one line, plus continuation lines (section 5.2).

- The fallback **SHOULD** be present and **SHOULD** be written as readable prose, because it is what plain renderers and failing renderers show (W101 if missing).

4.4 Inline components

An inline component `:name[fallback]{params}` appears within paragraph text, headings, list items and table cells.

- The fallback is **REQUIRED** for inline components, since the `[` is part of what makes the form recognizable.
- The fallback is parsed as inline Markdown and **MAY** contain other inline components.

4.5 Fallback rendering

When a CompMD processor cannot render a component (unsupported definition language, runtime error, missing import in lenient mode), it **MUST** render its fallback and report the matching diagnostic:

- a leaf or inline component: its fallback text;
- a code block a code renderer renders (section 7.7): the code block, as a plain renderer shows it;
- a container: its default slot, then each named slot in source order, as plain Markdown, so no slot's content is lost;
- an `if` whose `when` fails: both branches, as a container;
- an `each` that fails: its template once, as a plain renderer shows it.

A component on its way. A use of a component whose pack is pending (section 6.4) renders the same fallback, marked as pending: in the expanded tree it is a `componentPending` node around the fallback (section 12.3). It reports no diagnostic. A renderer **SHOULD** show that the content is loading, without hiding the fallback, and draws the component in its place when its pack arrives.

5. Parameters

Parameters are `key=value` pairs, checked against the component's signature. They never span lines except through continuation lines, so a broken quote or bracket damages only its own line.

5.1 Forms

Written	Means
`title="Setup"`	string `Setup`
`icon=rocket`	bare value; typed by the signature
`count=3`	number 3 if the signature says number, else string `3`
`open`	boolean `true`
`tags=["a","b"]`	JSON array
`meta={"id":1}`	JSON object
`.wide`	adds class `wide` (repeatable)
`#intro`	sets id `intro`
`md:gap=l`	`gap` override from breakpoint `md` up (section 9.3)

- A bare value is converted using the declared type: `number`, `boolean` (`true`/`false`), `string`, or an enum member. Conversion failure is F009.
- Each key **MAY** appear once per component invocation, counting continuation lines (F017 on a repeat).
- `class` and `id` are accepted by every component and need no declaration. `class` and `.name` add up: `{.person class=(p.team) .wide}` gives the classes `person`, the value of `p.team` and `wide`, in written order, without duplicates. `id` and `#name` set the same thing, so both together is F017. So are `lang` (a BCP 47 tag) and `dir` (`ltr`, `rtl`, `auto`), for a part in another language: `::: box {lang=fr}` around a French quote, or `dir=auto` around text from data whose direction isn't known. A code block a code renderer renders also accepts `alt`, its text alternative (section 7.7).
- A param or slot name may contain `-`. In expressions and interpolation paths, which follow CEL's identifier rules, it is written with `_`: inside a definition, `align-items` is read as `align_items`, and the slot `:: #side-panel` is placed with `{{md.slot.side_panel}}`. Two

params of one component, or two of its slots, whose names differ only in `-` and `_` are F013.

`name=(expr)` passes the result of an expression (section 8.2), typed by the expression. `{{path}}` is allowed in param values only inside quoted strings.

5.2 Continuation lines

A `::: + params` line adds params to the immediately preceding container opener, leaf, or continuation line. It takes the same param syntax without braces.

```
::: grid {gap=l align-items=start}
::: + md:grid-template-columns="2fr 1fr"
::: + lg:gap=2.5rem
```

A continuation line anywhere else is error F012.

5.3 Validation

A processor **MUST** validate every invocation against the signature: unknown keys (F008), type mismatches (F009) and missing required params (F010). Each diagnostic points at the param, not the whole line.

---

6. Imports and exports

Every `.comp.md` document can import components and export its own; a **pack** is simply a document (or published package) that exports components. Both are declared under the `md` frontmatter key, as `import` and `export`.

```
---
md:
  version: 1
  import:
    - from: "@myorg/ui@1"
      names: [card, callout]
    - from: "@other/kit"
      names:
        - name: card
```

```
          as: promo
    - from: ./chart.jsx
      names: [chart]
    - from: "@acme/icons"
      names: "*"
  export:
    pack: "@myorg/release-kit"
    version: "1.2.0"
    names: [callout, { name: chart, as: trend-chart }]
---
```

6.1 Imports

Key	Type	Rule
`md.import`	list of entries	optional; `@compmd/core` is always imported and never listed
`import[].from`	string	required; `@scope/pack`, `@scope/pack@<version>` with a semver version or range (`@1`, `@^1.4.2`, `@1.4.2`), or a relative path (`.comp.md`, `.js`, `.ts`, `.jsx`, `.tsx`, `.svelte`)
`import[].integrity`	string	optional; a Subresource Integrity hash of the pack (`sha512-…`); requires an exact version (F009 otherwise)
`import[].names`	list (at least one entry) or `"*"`	required
`names[]` as a string	name	imports that export under the same local name
`names[].name`	name	the export's name in the source
`names[].as`	name	the local name; defaults to `name`

- A component file (`.js`, `.ts`, `.jsx`, `.tsx`, `.svelte`) exports exactly one component, its default export, with the binding's default `runtime`, named after the file stem as written (`HabitHeatmap.jsx` exports `HabitHeatmap`). Names are never converted between cases; `as` renames. A stem that isn't a valid name needs `as`.
- `names: "*"` imports every export of the source under its exported name.

- Scoped names never appear in the body; `::: @myorg/card` is a malformed component line (F016).
- **Pinning.** A range takes the newest matching release, so a pack's next release changes what the document renders. A document that must not change pins an exact version with `integrity`; a processor **MUST** check the hash and refuse a pack that doesn't match (F035). Tools write the hash; it is the one the registry publishes (section 6.4). A pack pins its own imports the same way; a processor **MAY** warn when a pinned pack imports a range. The spec defines no lock file; a processor **MAY** keep one.

```
import:
  - from: "@acme/charts@1.4.2"
    integrity: sha512-9c1fQe…
    names: [chart]
```

6.2 Exports and pack identity

Key	Type	Rule
`md.export`	mapping	optional; a document without it exports nothing
`export.pack`	string `@scope/name`	optional; the pack's published name. Required only to publish to a registry
`export.version`	semver string	optional; the pack's version
`export.names`	list or `"*"`	required; strings or `{name, as}` mappings, where `as` is the public name

- An exported name **MAY** be a local definition or an imported local name, so packs can re-export.
- `names: "*"` exports every local definition, but not imports; re-exports are always listed.
- If the pack is also described by a registry manifest (such as `package.json`), `pack` and `version` **MUST** match it (F022).
- Importing a document by path needs no `pack` key; the path identifies it.

6.3 Resolution

A name in the body resolves through these sources, highest priority first:

1. Local definitions (`component=` fences in this document).
2. Explicitly listed imports.

3. Wildcard (`"*"`) imports and `@compmd/core`, at the same level.
4. Ambient packs a processor provides.

Rules:

- Two local names from levels 1 and 2 that are equal are F013, reported on the second declaration.
- A wildcard name equal to a level 1 or 2 name is silently shadowed, so a pack update can never change the meaning of a name you defined or listed.
- A name provided by two sources at level 3 (two wildcard imports, or a wildcard import and `@compmd/core`) is not an error until it is used; using it is F023, and listing it explicitly resolves it. So neither a pack update nor a new `@compmd/core` version can silently change what a name means.
- **Code blocks resolve by language** (section 7.7): a code block's language selects the code renderer, among the components the document can name, whose signature names that language, through the same levels. Two at the same level are F023 when a block of that language is used; listing one explicitly resolves it. Renaming a code renderer on import (`as`) renames it as a leaf only; it still renders its language.
- **Keywords.** `if`, `each`, `value` and `box` are fixed for a major version: no document may define, import, export or rename to them (F013).
- A local definition or explicit import with the name of another `@compmd/core` component shadows it, with W112. A name `@compmd/core` adds in a later version therefore never breaks a document that already uses it.
- Importing a name its source does not export is F021, reported on that `names` entry.
- Import cycles are F020, reported on the `from` that closes the cycle.
- **Scope is lexical.** An imported definition resolves names and `md.doc` in its own document, never in the importer's. A `compmd` definition's body is a document of its own too: it resolves footnotes and link reference definitions among its own lines, so a `[^1]` in a body doesn't reach a note outside it. Slot content resolves in the document it is written in: in `::: card` ... `::: /card`, the content's components, `md.doc`, `md.input` and `each` variables are the caller's, even though `card`'s body comes from another document.

Ambient packs. A processor **MAY** provide packs that every document imports implicitly, like `@compmd/core`, at the lowest priority, so local names and imports always win. Names from them are not portable: another processor reports F002 and renders fallbacks. A processor **SHOULD** also let documents import an ambient pack explicitly (`from: "@host/components"`), so a document can state what it depends on, and **MAY** report W113 for each ambient name used.

- **An ambient pack's name is the processor's.** A document that imports an ambient pack by name gets the processor's pack, never a registry package of the same name, as

for `@compmd/*` (section 6.4). So a package published under a host's pack name can't stand in for the host's components.

- An ambient pack **MAY** declare components the host implements, by their signatures alone (section 7.1).

Libraries a processor offers. A processor **MAY** also offer libraries that component code imports by their npm name (`import * as d3 from "d3"`, section 7.6). Like names from ambient packs, they aren't portable: on a processor that doesn't offer the library, the component renders its fallback with W121. A processor **MAY** report W113 for each component whose code imports one.

6.4 Packages and the registry

- **`@scope/pack` is the npm package of that name**, fetched from the npm registry or an npm-compatible registry the processor is configured with (a private registry, a mirror). A processor **MAY** resolve from a local cache or installed packages, so documents work offline. It **MAY** offer other registries with the same naming. What a pack on a registry must be, how a processor reads one, and the library packs packs share are in the companion *CompMD packs on a registry*.
- **The pack's document** is named by the `compmd` field of its `package.json` (`"compmd": "./kit.comp.md"`), by default `index.comp.md`. `@scope/pack/file` (in a data `from`, section 8.6) is that file in the package.
- **`package.json` is the registry manifest** that `export.pack` and `export.version` must match (F022). Its `integrity` is the hash an import's `integrity` key holds.
- **`@compmd/*` is reserved** by this spec and never fetched from a registry: processors build `@compmd/core` in, whoever holds that scope on a registry.
- Names are always scoped; an unscoped `from` that isn't a relative path is F026.
- An import whose source can't be loaded (a missing file, no release matching the version) is F026. Its names are dropped, and their uses render fallbacks without F002.
- **Pending packs.** A processor whose host gets packs while a reader views the document (from a registry, in an editor) **MAY** render before a pack arrives. Its host then names the packs, and the data sources (section 8.6), it is still getting: they are *pending*. A pending import reports nothing: its names stand for pending components, whose uses render their fallbacks marked as pending (section 4.5). When the pack arrives, the document renders again (section 8.5). The processor never decides that a pending source failed; the host does, on an error from the registry or after a time it chooses, and processes again without it among the pending: an import that can't be loaded then, F026. A processor that never fetches while it renders has no pending sources.
- **Which packs a document may import** is the processor's configuration. By default, any pack in the registry: its code runs in the strict sandbox (section 15), so a pack nobody reviewed can't reach the reader's data, and only documents that import it are affected. A

processor **MAY** instead allow only the packs its configuration lists, and **MAY** deny packs by name or version. A pack it doesn't allow is F026, with the reason in the message. A deny list is the host's own policy (content it won't show, packs it won't serve, licences, misuse); it isn't what keeps readers safe, which the sandbox does for every pack, known or not.

- **The processor gets packs.** A processor that renders for a reader fetches packs itself or through its own cache, so the registry doesn't learn who reads which document (section 15, "Privacy"). Its configuration **MAY** instead let the reader's browser fetch them, from the registries the configuration names; it **SHOULD** then tell its readers, since the registry learns which packs they load (*CompMD packs on a registry*, section 4). A document can never choose this. A processor **MAY** cap a pack's size; a larger pack is F026.
- **Built modules.** A component file in a pack **MAY** be a built ES module (`.js`, the plain binding of section 7.6) with every library it imports bundled in, but its binding's runtime and the libraries its pack names in `peerDependencies`, which the processor provides (section 7.6). A built module **SHOULD** be one file, with no relative imports: a processor that passes a built module's text to where it runs can't load one that imports files of its own, and renders its components' fallbacks with W121. A processor **MAY** load such a file where it runs the component, by its place in the pack (the pack, its version and the file's path), instead of passing its text: the file **MUST** be the one the pack published, from files checked against the pack's integrity when the processor got them (F035). Loaded once, it can be cached by its place, since a published version never changes.
- **Compiled ahead.** A processor **MAY** compile a pack's source component file (JSX, TSX, Svelte, TypeScript; section 7.6) once, before a document needs it, from the file the pack published, and load the result where it runs the component, by the file's place, instead of compiling its text there. The result is the same code compiling it there would give; it runs at its pack's trust, as its source would, and can be cached by the place, as a built module can.
- **However a processor gets a registry pack, it is a registry pack.** One the host installed with an application, one it keeps in its own store of published versions (a marketplace), and one fetched while a reader views the document are the same pack: its files are checked against its integrity, its trust is what the configuration gives it, never trusted (section 15), and caching by its place holds for each.
- **What a pack needs from the processor** is the `peerDependencies` of its `package.json`: its binding's runtime with the lowest version it works with (`"svelte": "^5.40"`), and the libraries it expects the processor to provide (`"d3": "^7"`), which the processor offers or gets from a registry as library packs (*CompMD packs on a registry*, section 5). A processor checks them when it resolves the pack, against the runtimes and libraries it has (section 7.6): one it doesn't have, or has at a version outside the range, is W121 on the import, naming it, the range and the version it has. The import is kept: the pack's components that don't need it still render, and those that do render their fallbacks with W121.

7. Defining components

A component is defined in a fenced code block whose info string is the source language followed by `component=<name>`, the language one of the definition languages in the table below. Plain renderers show it as highlighted code; CompMD processors remove it from the rendered output. The first word is always the source language, so plain renderers can highlight it; anything else goes in attributes after it. A code block in any other language is content, even with `component=` in its info string.

Language	`runtime` (default first)	Defined in
`compmd`	none, every processor runs it	section 7.2
`jsx`, `tsx`	`react@19`, or `preact@10`	the React binding (section 7.6)
`svelte`	`svelte@5`	the Svelte binding (section 7.6)
`js`, `ts`	none: plain JavaScript	the plain binding (section 7.6)

- `runtime=<framework>@<major>` names the framework a host-language component is written for: ```` ```svelte component=card runtime=svelte@5 ````. Without it, the binding's default applies. An **exported** host-language component without `runtime` is W117, since the default may move to a later major version. Plain JavaScript has no `runtime` and no W117; a `js` or `ts` component with a `runtime` names a runtime no processor has (W102).
- Host-language components run through the host contract (section 7.4); a **binding** maps it to one framework. Version 1 has bindings for plain JavaScript, React and Svelte 5; others (Vue) come in minor versions. A processor without the binding or runtime renders fallbacks (section 4.5) and reports W102.
- **The component's box.** A definition fence **MAY** end with a param list, which styles the component's box (section 9.2), as `:host` styles a custom element: ```` ```compmd component=card {.card padding=s md:padding=m} ````. It takes the common params, `class`, `id`, `lang` and `dir`, and on a `compmd` fence also `box`'s own params (`display`, `gap`, ...). Its expressions are evaluated for each invocation, in the definition's scope: a `compmd` definition's params, `md.doc` of its own document and `md.data`. Breakpoints pass through as in the body (section 9.3). A signature fence takes no param list (F036).
- A definition fence **MAY** appear anywhere at document level; the name is visible to the whole document.
- An imported `.comp.md` file **MAY** hold several definitions; it offers those listed under `md.export`.

- A definition whose body contains a fenced code block uses a longer outer fence, or `~~~`, as CommonMark requires: a fence closes only at a fence of the same character that is at least as long.

````
````compmd component=snippet
params:
  code: string
---
```sh
{{code}}
```
````
````

7.1 Host-language definitions

````
```svelte component=card runtime=svelte@5
<script>
  let { title, icon = "info", children, footer } = $props();
</script>
<section class="card"><h3>{title}</h3>{@render children?.()}<footer>
{@render footer?.()}</footer></section>
```
````

Signatures. A host-language component's signature (params, types, defaults, slots) comes from a `compmd` **signature fence**, or is derived from the component:

````
```compmd component=card signature
params:
  title: string
  icon: info | rocket | warning = info
slots: [default, footer]
```
````

- A signature fence uses the `compmd` language with the `signature` flag, holds only the signature (section 7.3; no `---` or body), and attaches to the component of that name, whether defined in the same document or imported from a component file.
- **Components the host implements.** In an ambient pack (section 6.3), a signature fence with no component of its name declares one whose code the processor provides, such as a component built with the host application. Its uses are validated against the signature and rendered through the host contract with the processor's code, at the trust level its

configuration gives it (section 15). In any other document, a signature fence with no component of its name is F002.

````
```compmd component=stars signature
params:
  value: number
```
````

- **Exported** host-language components **MUST** have a signature fence (F029). Packs are used by processors that may not run the component, and still need validation and fallbacks.
- **Local** host-language components **MAY** omit it. The processor then derives what it can: TypeScript prop types, Svelte `$props()`, Vue `defineProps` / `defineSlots`, destructuring defaults, and node or snippet props as slots. Anything not derivable is `dyn` and passed through unchecked, with W105.
- When a signature fence is present it is authoritative. A processor that can analyse the component **SHOULD** check that it conforms and report W104 for any mismatch (for example a signature param with no matching prop).

7.2 The CompMD definition language

A `compmd` definition has a signature header, a `---` line, and a body of CompMD. It needs no host framework, so it renders in every CompMD processor.

````
```compmd component=card {display=flex flex-direction=column gap=s
padding=m border=subtle border-radius=m class=(tone)}
params:
  title: string
  icon: string = "info"
  tone: info | warn = info
slots: [default, footer]
---
# {{title}}
{{md.slot}}

{{md.slot.footer}}
```
````

A block invocation of a `compmd` component is one box, styled by the fence's params, holding the body (section 9.2).

An inline invocation of a `compmd` component (`:badge[beta]{tone=info}`) puts the children of its body's paragraph in the text; a body that isn't a single paragraph renders the fallback. It has no box, so the fence's params don't apply.

Headings in a body are relative by default: `#` is one level below the heading the invocation is under, `##` two levels below, and so on, up to level 6; with no heading above the invocation, `#` is level 1. So `card`'s title is an h3 under `## What changed` and an h2 under the document's `#` title, and the outline stays correct wherever `card` is used. Slot content isn't shifted: the caller wrote its levels. A definition whose headings must keep their written level (a title page) declares `headings: absolute` in its signature (section 7.3). The parse tree keeps the written levels; the expanded tree has the computed ones (section 12.3).

7.3 Signatures and type declarations

A signature (the header of a `compmd` definition, or a signature fence) is YAML: `params` maps each param name to a declaration, and `slots` lists slot names. The same declarations are used for `inputs` (section 8.7) and, with `from`, for a data import's `schema` (section 8.6).

A declaration has a short form, a YAML string in this grammar:

```
type-decl = type [ "?" ] [ 1*WSP "=" 1*WSP default ]
type      = "string" / "number" / "boolean" / "date" / "array" / "object"
          / "length" / "space" / "color" / "tracks" / "areas" / "surface" /
"border" / "radius"
          / "name" / enum / input
enum      = name 1*( *WSP "|" *WSP name )
input     = "input<" ( type / "enum" ) ">"          ; a bound input
(section 7.4)
default   = value                                   ; the param value
grammar, section 3.1
```

and a long form, a YAML mapping for what the short form can't say well:

```yaml
params:
  tone: info | warn = info              # short form
  icon: string?
  meta:                                 # long form
    type: object
    default: {a: 1}
    doc: Extra data passed to the chart
```

Key	Rule
`type`	required; a `type` from the grammar above, with an optional `?`
`default`	optional; any YAML value of that type
`doc`	optional; a description for editors and tools
`from`	data `schema` only; the key in the data, when it differs from the declared name

- A param without a default is required, unless its type ends in `?` (optional, no value).
- A type that isn't in the grammar, such as a misspelt `numbr`, is F037, reported at the type name, inside `input<…>` too. The declaration is kept, but its type is `dyn`: it takes any value and checks none, so the one mistake gives one diagnostic, not one at each use. A single name is never an enum, which has at least two members.
- `input<enum>` binds an input of any enum type, whose members the component reads as its choices (`select-input`, section 8.7); `input<a | b>` binds only an input of that enum.
- Enum members are names, written without quotes, so every short form is a valid YAML string. A member with spaces needs the whole declaration quoted (`status: '"to do" | done = done'`).
- A short form whose default contains `:` (a JSON object, some strings) isn't valid YAML; use the long form.
- `slots` lists slot names; `default` names the default slot. A component with no `slots` key takes no body: using it as a container is F018. `slots` may instead be a mapping from slot name to options, for template slots (section 7.5): `slots: {default: {each: rows, as: as}, footer: {}}`.
- `headings` is `relative` (the default) or `absolute`: whether headings in a `compmd` definition's body are shifted to fit where it is used (section 7.2).
- `renders` is a language, or a list of languages that are names of one (`[mermaid, mmd]`): the component is also a code renderer for them (section 7.7). It has a required `string` param `code`, which holds the block's text, without declaring it. A signature **MAY** declare `code` to document it, but only as a required `string` with no default (F009 otherwise). A code renderer has no `slots`.
- A `name` param is a bare value that must be a CEL identifier and is never evaluated: the name a template slot binds. A `name` param given an expression or anything but an identifier is F009. There is no `expression` type: any param accepts `(…)`, and its type says what the value is, not how it's written.
- A `compmd` definition's body **MAY** invoke other components, including built-ins. Direct or indirect recursion is F019. A code block a code renderer renders invokes it (section 7.7), so a body that leads back to its component through such a block recurses too.

7.4 The host contract

A processor that renders HTML runs a host-language component through one contract, whatever its framework. Processors for other outputs render its fallback (W102).

```
mount(target: Element, context: {
  params: Record<string, unknown>,        // validated against the
signature, typed, with the defaults of those not given
  slots: Record<string, SlotItem[] | SlotInstance[]>, // each slot's
content, by slot name ("default" too)
  set(param: string, value: unknown): void, // changes the input bound to
an `input` param
  fallback: { text: string; element: Element }, // the invocation's
fallback, as plain text and rendered
  headingLevel: number,                   // the level the component's top
heading should have (section 7.2)
  scheme: "light" | "dark",               // the colour scheme the
component is shown in (below)
  fail(message: string): void,            // the component can't draw what
it was given (below)
  busy(work: Promise<unknown>): void,     // the component is still
drawing, until `work` settles (below)
  host?: unknown,                         // what the processor gives
components of permissive and trusted packs (below); absent otherwise
}): { update?(params, slots): void; destroy?(): void }

type SlotItem = { element: Element; component?: { name: string; params:
Record<string, unknown>; pending?: true } }
type SlotInstance = { value: unknown; items: SlotItem[] }   // a template
slot: one per element (section 7.5)
```

- **The outer box is the processor's.** It creates `target`, applies the common params and `class`, `id`, `lang` and `dir` to it (the definition fence's, overridden by the invocation's), and lays it out (section 9.2). The component owns what is inside `target` and doesn't receive those params.
- **Slots are rendered by the processor**, into elements the component places in its own DOM. Components inside a slot are mounted by the processor and stay live, whatever framework the parent uses.
- **A slot is a list of items**, one per top-level block (section 9.1): its element, and, when the block is a component, its name and params. A tabs component reads its tabs' labels from the items' params. A use of a component still on its way (section 4.5) is an item whose `component` has its name, no params, and `pending: true`. An item's element is a box the

processor makes around the block, so a block placed by position (section 9.2) is placed in its item, whatever the component's own markup; the component sizes it.

- **An item's width is the component's to give.** An item is the nearest box of the block in it, so when that block uses breakpoints, the item is the box they follow and doesn't take its width from its content (section 9.3). A component that lays its items out by their content, such as a toolbar that puts them side by side, gives such an item a width. A processor can't check this, since only the component knows its layout.
- **Every slot is rendered up front**, including content the component hides; hiding is the component's own styling. A component may be laid out while hidden or at zero width, and **SHOULD** lay itself out again when it's resized.
- **Updates, not remounts.** When params or slots change (an input or data changes, section 8.5), the processor calls `update`; the component keeps its state. `destroy` is called before `target` is removed.
- **The colour scheme.** `scheme` is the colour scheme the processor shows the component in: `"dark"` when the box it renders into is dark (in HTML, its `color-scheme`, and the reader's preference where it allows both), `"light"` otherwise. When it changes (the reader's preference, the host's own theme switch, printing), the processor sets `scheme` and calls `update` with the same params and slots. A component that takes its colours at render, such as a diagram or a canvas chart, reads `scheme`, not the reader's preference, which may differ from the page's. A component styled with CSS needs neither.
- **A component that fails.** A component that finds it can't draw what it was given after `mount` or `update` returned (an asynchronous parse, a module loaded late) calls `fail(message)`. The processor reports W118 with `message`, and shows the fallback in `target`; until the next `update`, `target` is the processor's. Before that `update`, the processor empties `target` again, so a failure lasts until the params, the slots or the scheme change. `fail` is for input the component can't draw, not for a resource that may come back: a map whose tiles stop loading shows its own state and retries, rather than replacing a usable map with its fallback.
- **A component still drawing.** A component that draws after `mount` or `update` returned (a diagram drawn asynchronously, data it gets through `host`) calls `busy(work)` with a promise of that drawing, once for each. A processor that waits for its page to settle, to print it, to scroll to a link's target, or in a test, waits until every such promise has settled, fulfilled or rejected, or as long as it chooses. It shows nothing differently while a component is busy. A component that never calls `busy` has drawn when `mount` or `update` returns.
- **A component that throws** while mounting renders its fallback, with W118, and is never mounted. One that throws while updating is treated as if it called `fail`: it is still mounted, and is updated again at the next change.
- **What a host gives permissive and trusted components.** A processor **MAY** give components from packs its configuration makes permissive or trusted (section 15) more than this contract, in `host`: an API to the host's data, events to the host. What `host` holds

is the processor's, and it is the same at both levels, so a component written against it works whichever of the two its pack is given; a trusted component reaches what the processor's own code reaches anyway, so `host` gives it nothing more than a way to be written once for both. The trust level comes from the processor's configuration, never from a document. A document that uses such components works only in that host, as with names from ambient packs (section 6.3). Section 15 says what such components may do with it.

- **Accessible text.** A component that renders non-text content (a chart, a map, a canvas) **SHOULD** use `fallback.text` as its text alternative, for example `<svg role="img" aria-label="…">`, since the leaf's fallback is the sentence plain renderers show in its place. Only the component knows the role of its content, so the processor doesn't label `target`.
- **Changing an input.** A param declared `input<type>` (section 7.3) is bound to an input by the document (`selected=region`): the component receives the input's current value, and `set("selected", value)` changes the input, checked against its type, after which the processor renders again (section 8.5). `set` on any other param, or during `update`, is refused. Components have no other outputs in version 1.

```
:: chart[Sales by region] {rows=(md.data.sales) selected=region}
:: table[Sales] {rows=(md.data.sales.filter(s, md.input.region == "" ||
s.region == md.input.region))}
```

Clicking a bar calls `set("selected", "eu")`; the table renders again with the rows for `eu`.

7.5 Template slots

A template slot is rendered once per element of an array param, with a name the author picks for the element. It lets a component decide where each record's template goes: a board that groups cards by status, a calendar that puts each event in its day.

```
# the board's signature
params:
  rows: array
  by: string
  as: name
slots:
  default: {each: rows, as: as}
```

```
::: board {rows=(md.data.tasks) by=status as=t}
**{{t.title}}**, due {{t.due}}
```

```
::: /board
```

- `each` names an `array` param and `as` a `name` param of the same component. The processor renders the slot once per element, up front, with the element bound to the name the author gave (`t`) and `md.loop` bound as in `each` (section 8.3).
- The name is chosen at the call site, so the slot content resolves it like any of the caller's names (section 6.3).
- A host component receives the slot as a list of instances, `{value, items}`: the element and its rendered template (section 7.4). In a `compmd` definition, `{{md.slot}}` places all instances in order.
- One template slot iterates one array. A table with a template per cell is a component that takes `rows` and `cols` and renders cells itself.
- A computed value per element is computed in the data first, with CEL's `map`: `rows=(md.data.tasks.map(t, {"task": t, "lane": t.due < md.today() ? "late" : t.status})) by=lane`.
- Plain renderers show the template once, like `each`'s (section 14).

To repeat a component once per record in order, no template slot is needed: put the component inside `each` (section 8.3).

7.6 Bindings

A binding maps the host contract to one framework.

Binding	**Language, `runtime`**	**Default slot**	**Named slots**	**Items, instances, `set`, `fallback`, `headingLevel`, `scheme`, `fail`, `busy`, `host`**
React	`jsx`, `tsx`; `react@19`, `preact@10`	`children`, a node that places the slot's elements	a prop of the same name, the same kind of node	a `compmd` prop: `{slots, set, fallback, headingLevel, scheme, fail, busy, host}`
Svelte	`svelte`; `svelte@5`	the `children` snippet	a snippet prop of the same name	a `compmd` prop: `{slots, set, fallback, headingLevel, scheme, fail, busy, host}`
Plain	`js`, `ts`; none	`slots.default`	`slots.<name>`	the context, as in section 7.4

In the React and Svelte bindings, params are props of the same name (a param with `-` in its name becomes camelCase: `align-items` is `alignItems`; so does a named slot's prop: the slot `side-panel` is `sidePanel`), and the binding implements `mount`, `update` and `destroy` with the framework's own mounting. In the plain binding the default export *is* `mount`, with the context of section 7.4, and params keep their names:

```
export default function mount(target, { params, fallback }) {
  const chart = drawChart(target, params.rows, fallback.text);
  return { update: (next) => chart.update(next.rows), destroy: () =>
chart.remove() };
}
```

Runtimes. A binding's runtime is the framework at the major version its `runtime` names: React 19 with React DOM, Preact 10, Svelte 5. The processor provides it, one copy for all the components it runs together (two copies of React break its hooks), so a component's code imports it by the framework's own module names (`react`, `react/jsx-runtime`, `react-dom/client`, `svelte`, ...) and never bundles it. Which minor version a processor has, and which of the framework's modules, is the processor's to say, in the runtimes and libraries it states it has (its configuration); a processor **SHOULD** keep its runtimes at a recent minor version. A processor without a runtime renders its components' fallbacks with W102 (section 7).

What component code may import. A host-language component's code, in a fence, in a file or built, imports only:

- its binding's runtime;
- a library the processor offers (section 6.3), by its npm name;
- a library its pack names in `peerDependencies`, which the processor provides (section 6.4);
- in a built module, its own files in its pack, by relative path.

Anything else resolves nowhere. Such an import, a compile error, and a built module whose file doesn't match its hash are W121: the component renders its fallback.

Compiling source. A processor that runs a component from its source compiles it as its binding says:

- **React:** JSX and TSX with React's automatic runtime, imported from `react/jsx-runtime`; TypeScript's types removed.
- **Svelte:** with Svelte 5's compiler, for the client.
- **Plain:** `js` runs as written; `ts` has its types removed.

Imports in source are bare names, as above. A relative import in source (`./helpers.js`) is W121, since source has no place in a pack to resolve from. Where and how a processor

compiles (in a sandbox, on a server, ahead of time) is its own choice.

7.7 Code renderers

A **code renderer** is a leaf whose signature also names a language with `renders` (section 7.3): it renders the code blocks of that language in documents that can name it, as well as being invoked as a leaf. A document writes the block as it would anywhere, so a plain renderer shows the code, and GitHub draws a Mermaid diagram itself.

````
```compmd component=mermaid signature
renders: [mermaid, mmd]
params:
  theme: default | dark | neutral = default
```
````

````
---
md:
  import:
    - from: "@acme/diagrams@1"
      names: [mermaid]
steps: [Draft, Review, Published]
---

```mermaid {theme=dark alt="Draft, then review, then published"}
graph LR; Draft graph LR; Draft --> Review  Review --> Published
```

:: mermaid[Our process: draft, review, published] {code=("graph LR; " +
md.doc.steps.join("  + md.doc.steps.join(" --> "))}
````

- **Which form.** Source in a language that exists outside CompMD, which a reader of the plain Markdown should see as source, is a code block. Source computed from data or inputs is the same component as a leaf, its source as its `code` param, since a code block is never interpolated (section 8.1).
- **The block's text** is the `code` param. The block's info string gives the other params, as a param list after the language (section 3); giving `code` there too is F017. The component can't tell the two forms apart, and the host contract is the same (section 7.4). It isn't told the block's language: the names in a list are names of one language.
- **Resolution** is by the block's language (section 6.3). A block whose language no code renderer renders stays a code block, with no diagnostic, since most code blocks are just code.

- **A box.** A rendered block has an outer box, as a leaf does, so the common params apply (section 9.2).
- **The text alternative** is `alt`, which every rendered block accepts, as every component accepts `class` and `id`; it is the component's `fallback.text`. A rendered block without `alt` is W101, as a leaf without fallback text is.
- **The fallback** is the code block itself (section 4.5).
- **Blocks only.** CommonMark gives a code span no language, so a code renderer renders no inline code.
- Math is not a code renderer: a ```` ```math ```` block is display math in every processor (section 2.3), and a code renderer for `math` is never used.

---

8. Interpolation, expressions and control flow

8.1 Interpolation

`{{path}}` inserts a value. It works in document bodies and in `compmd` definition bodies, and values come only from the names in this table.

Path	Available in	Value
`md.doc.<key>`	everywhere	an author's frontmatter key (section 2.1)
`md.version`, `md.lang`, `md.tz`	everywhere	the document's version, language and time zone (section 2.1)
`md.data.<name>`	everywhere	a data import (section 8.6)
`md.input.<name>`	document bodies	an input's current value (section 8.7)
`<param>`	`compmd` definition bodies	the invocation's param value or its default
`<as name>`	inside `each` and template slots	the current item (sections 7.5, 8.3)
`md.loop.<field>`	inside `each`	iteration info (section 8.3)
`md.slot`	`compmd` definition bodies	the default slot's rendered content

Path	Available in	Value
`md.slot.<name>`	`compmd` definition bodies	that named slot's rendered content, or empty

Everything CompMD provides, variables and functions, is under the `md.` namespace; other names are chosen by authors (params, `as` names and inputs). Authors cannot declare the name `md` (F013).

Form. `{{`, optional spaces, a path, optional spaces, `}}`, on one line. A path is names joined by `.`, each a CEL identifier (`[A-Za-z_][A-Za-z0-9_]*`); v1 has no expressions, operators, indexes or filters. A param or a slot named with `-` is written with `_` (section 5.1); data keys that aren't identifiers are renamed in the data import's `schema` (section 8.6).

Where.

- In text, in fallback text, in link and image destinations and link reference definitions, in quoted param strings (`title="Step {{n}}"`), and in data `from` values (section 8.6). Unquoted param values take expressions instead (section 8.2).
- Interpolation is recognized like a code span: before emphasis and links, so `*{{a*}}` is not an interpolation.
- Never in code spans or code blocks, except in `compmd` definition bodies.
- Never in raw HTML: `<img src="{{md.doc.logo}}">` stays literal. Use Markdown instead, `![logo]({{md.doc.logo}})`, which gets `urlParts` and the URL scheme rule below.

What it inserts.

- In text, the value as text, never re-parsed as Markdown; the renderer escapes it for its output. Slot values are inserted as rendered content.
- Values that aren't strings are written as in this table. Formatting for readers (separators, currencies, date styles) is always explicit, with `:value` and a formatting function (section 8.4).

Value	Written as
number	as JavaScript's `String(n)` writes it: the shortest form that reads back as the same number, `12.5`, `3`, `0.1`; with an exponent below 10^{-6} and from 10^{21} up, `1.5e-7`, `1e+21`; and `NaN`, `Infinity`, `-Infinity`
boolean	`true`, `false`
`null`	nothing
date	as stored: `2026-10-21`

Value	Written as
timestamp	RFC 3339 in UTC: `2026-10-21T10:00:00Z`
map with a string `name`	its `name`: a record that stands for something (a person, a linked record) reads as its name, `{{md.doc.author}}` as `Ursula K. Le Guin`
list, other map	not written: F024. Use `each`, or `:value` with `join`

- In link and image URLs, characters not allowed in a URL are percent-encoded; `/`, `?` and `#` are kept. Interpolation never produces a URL's scheme: when a URL starts with an interpolation, the result **MUST** be a relative URL or use `https:`, `http:` or `mailto:`. Otherwise the link's text is rendered without the link, with W109.
- In quoted params, the value as text inside the string.
- Processors that make heading anchors use the expanded text.

Errors.

- An unknown path is F014. Its message suggests `\{{` or a code span, since prose that mentions `{{name}}` is read as an interpolation.
- `{{` followed by something that isn't a path (`{{ 1 + 2 }}`, `{{a b}}`) is literal text, with W106.
- `\{{` writes literal `{{`.

A `release.comp.md` file, CompMD by its extension, with only the author's frontmatter:

```
---
version: "1.4"
---

Install RayPlay {{md.doc.version}} with `cargo install rayplay`.
```

8.2 Expressions

A param value written in parentheses, `name=(expr)`, is an expression in the **CompMD CEL profile**, a defined subset of the Common Expression Language. Computation happens only in expressions on component lines, never in prose.

```
:: badge[{{status}}] {tone=(status == "done" ? "success" : "neutral")}
::: box {class=("callout-" + kind)}
```

Grammar. The `value` rule in section 3.1 gains `expr = "(" cel-expression ")"`. The expression ends at the matching `)`, counting parentheses outside CEL string literals, and **MUST** fit on one line.

Variables. An expression sees the component's params (inside a `compmd` definition body), `md.doc` (the author's frontmatter of the document the expression is written in), `md.version`, `md.lang` and `md.tz` (section 2.1), `md.data` (section 8.6), `md.input` (section 8.7), and inside `each` the `as` name and `md.loop`. Nothing else exists: there are no environment, device or network variables. `md.doc`, `md.data`, `md.input` and `md.loop` are qualified CEL identifiers, declared to the CEL environment as variables next to the `md.` functions.

Types. CompMD types map to CEL types as follows; expressions are type-checked against the declared param type.

CompMD type	CEL type	Notes
`string`, enum	`string`	enum values are checked against the enum's members
`number`	`double`	also when the value has no fraction (`3`)
`boolean`	`bool`	
`date`	`string`	an ISO 8601 calendar date, `"2026-10-21"`
`array`	`list`	
`object`	`map(string, dyn)`	
`length`, `space`, `color`, `tracks`, `areas`, `surface`, `border`, `radius`	`string`	only `==`, `!=` and string concatenation are allowed; grammars in section 9.4

Numbers. Every number from frontmatter, data, params and inputs is a CEL `double`. Integer literals, `size()`, `md.loop` fields, `md.daysBetween` and list indexes are `int`, as in CEL. The profile adds arithmetic and comparison between `int` and `double`, which gives a `double`: `p.price * 12` and `md.input.rate * 100 > 5` need no conversion. Everything else follows CEL: dividing two `int`s truncates (`size(rows) / 2` with 3 rows is `1`; write `double(size(rows)) / 2`), and a list index must be an `int` (`rows[int(md.input.page)]`). `string()` writes a `double` as interpolation does (section 8.1): `string(1e21)` is `"1e+21"`. Map keys of different number types match by value, as in CEL: `{1: "a"}[1u]` and `{1: "a"}[1.0]` are `"a"`, so `md.groupBy` on a field of whole numbers from data gives `int` keys.

Dates and times. A `date` is a string holding an ISO 8601 calendar date (`"2026-10-21"`), so dates from frontmatter, data and inputs need no conversion, and they compare correctly with `<`, `>=` and `==`. Times are CEL `timestamp`s and `duration`s, for the few documents that need a time of day. Data holding full timestamps (`"2026-10-21T10:00:00Z"`) compares with a date as strings, which works for ranges in one zone; for exact times, convert with `timestamp(s.at)`.

Profile. The profile is exactly:

- **From CEL's standard definitions:** the operators, the conditional `?:`, the macros `has`, `all`, `exists`, `exists_one`, `filter` and `map`, and the functions `size`, `in`, `contains`, `startsWith`, `endsWith`, `int`, `double`, `string`, `timestamp` and `duration` with their `timestamp` and `duration` operations: arithmetic, comparison and the getters. A timestamp's getters (`getFullYear`, `getMonth`, `getDate`, `getDayOfMonth`, `getDayOfWeek`, `getDayOfYear`, `getHours`, `getMinutes`, `getSeconds`, `getMilliseconds`) read it in the time zone given, an IANA name or an offset such as `"+02:00"`, and without one, unlike CEL's UTC, in `md.tz`: `md.now().getHours()` is the hour of the render where the document is read. A duration's (`getHours`, `getMinutes`, `getSeconds`, `getMilliseconds`) count its whole units.
- **From CEL's strings extension:** `charAt`, `indexOf`, `lastIndexOf`, `lowerAscii`, `upperAscii`, `replace`, `split`, `substring`, `trim` and `join`.
- **The CompMD functions** in the tables below.

It excludes protocol buffer messages, `matches` (regular expression dialects differ between implementations, and some can be made slow), and every other extension.

Function	Example	Result
`md.currency(n, code)`	`md.currency(price * 10, "USD")`	`"$120.00"`
`md.number(n, digits)`	`md.number(0.183 * 100, 0)`	`"18"`
`md.date(d, style)`	`md.date(md.doc.released, "long")`	`"October 21, 2026"`
`md.plural(n, one, other)`	`md.plural(count, "item", "items")`	`"items"`
`md.plural(n, forms)`	`md.plural(n, {"one": "файл", "few": "файла", "many": "файлов", "other": "файла"})`	the form for `n`
`md.today()`	`md.today()`	the render date, a `date`

Function	Example	Result
`md.now()`	`md.now()`	the render time, a `timestamp`
`md.dateOf(t)`	`md.daysBetween(md.dateOf(md.doc.created), md.today())`	the `date` of a `timestamp` in the render's time zone (`md.tz`)
`md.addDays(d, n)`	`md.addDays(md.today(), -7)`	a `date`
`md.daysBetween(a, b)`	`md.daysBetween(md.today(), md.doc.launch)`	an `int`, negative when `b` is before `a`
`md.startOf(d, unit)`	`md.startOf(md.today(), "week")`	a `date`; `unit` is `"week"`, `"month"` or `"year"`

List function	Example	Result
`md.sort(list)`	`md.sort(md.doc.tags)`	ascending
`md.sortBy(list, field)`	`md.sortBy(md.data.sales, "amount")`	ascending by that field
`md.reverse(list)`	`md.reverse(md.sortBy(md.data.sales, "amount"))`	reversed
`md.slice(list, start, end)`	`md.slice(rows, 0, 5)`	the elements from `start` up to `end`
`md.sum(list)`	`md.sum(md.data.sales.map(s, s.amount))`	a number; `0` for an empty list
`md.min(list)`, `md.max(list)`	`md.max(md.data.sales.map(s, s.amount))`	a number, or `null` for an empty list
`md.unique(list)`	`md.unique(md.data.sales.map(s, s.region))`	first occurrences, in order
`md.groupBy(list, field)`	`md.groupBy(md.data.sales, "region")`	a map from each value to its elements, in order

List function	Example	Result
`md.list(list)`, `md.list(list, "or")`	`md.list(md.doc.contributors)`	`"Ana Lima, Kenji Sato, and Priya Nair"`, joined by the locale's list pattern (`"or"`: alternatives)

- List functions name a field instead of taking an expression, because macros that take expressions aren't portable across CEL implementations. To sort by a computed key, `map` it into a field first (section 7.5).
- Sorting is stable. Strings sort by code point, not by locale, so output does not depend on locale data.
- `md.list` writes each element as interpolation writes a value (section 8.1), so a list of records with a `name` gives their names: `md.list(md.doc.authors)` is `"Ana Lima and Kenji Sato"`.
- `md.today()` and `md.now()` read the render context (section 8.5): every call within one render returns the same value. `md.today()` is the date in the render's time zone (`md.tz`).

Locale. The formatting functions (`md.currency`, `md.number`, `md.date`, `md.plural`, `md.list`) and the first day of the week in `md.startOf` follow `md.lang` (section 2.1), which defaults to `en`, with the locale data of a recent CLDR release. The spec doesn't pin a release, so processors can use their platform's data; output in other languages may differ slightly between processors. `md.date` takes CLDR's styles `"short"`, `"medium"`, `"long"` and `"full"`. `md.plural` with a map picks CLDR's plural category for `n` (`zero`, `one`, `two`, `few`, `many`, `other`), falling back to `other`, which is required; the three-argument form is a shorthand for `{"one": one, "other": other}`.

Checking. Expressions in `compmd` definitions **MUST** be type-checked when the definition is loaded; expressions in document bodies are checked against the frontmatter when rendered. Any parse, type or evaluation error is F024, reported with the column inside the expression and, for unknown variables, the nearest declared name.

8.3 Control flow

Control flow is two ordinary `@compmd/core` containers, `if` and `each`. They use the same syntax, named closers and diagnostics as every other component; there are no keywords.

`if`

Param	Type	Meaning
`when`	`boolean`	required; the default slot renders when true

Slot `else` renders when `when` is false.

```
::: if {when=(kind == "warning")}
**Read before upgrading.**
:: #else
This release is backward compatible.
::: /if
```

Prefer an expression in a param over an `if` when only a value differs: `tone=(done ? "success" : "neutral")` instead of two branches.

`each`

Param	Type	Meaning
`in`	`array`	required; the items
`as`	`name`	required; the item's variable name, a CEL identifier (no `-`)
`join`	`none | list | inline = none`	how iterations combine
`sep`	string = `" "`	separator for `join=inline`
`last`	string?	separator before the last item for `join=inline`; defaults to `sep`

Slot	Renders
default	once per item, with the `as` variable and `md.loop` bound
`before`	once, above the result, only if there is at least one item
`empty`	once, only if there are no items

`md.loop` has `index` (0-based, matching CEL list positions), `number` (1-based, for display), `count`, `first` and `last`. The `as` variable and `md.loop` are visible in the default slot only;

nested `each` blocks see their own `md.loop`, and outer items by their `as` names.

Join modes. CompMD parses before it expands, so each iteration's default slot is already Markdown blocks; `join` says how they combine.

`join`	Each iteration's default slot must be	Result
`none`	anything	blocks appended in order
`list`	list items of one list type	one list. Ordered lists number continuously. Tight unless any item contains blank lines
`inline`	exactly one paragraph	one paragraph, iterations separated by `sep`, the last pair by `last`

An iteration whose shape does not match is F025, reported at the opener and naming the first offending line. A default slot that is a single list item with `join` unset is W103.

```
::: each {in=(md.doc.changes.filter(c, c.type == "breaking")) as=c
join=list}
- {{c.text}} ([#{{c.pr}}](https://github.com/acme/rayplay/pull/{{c.pr}}))

:: #before
## Breaking changes
::: /each

::: each {in=(md.doc.stack) as=s join=inline sep=", " last=" and "}
[{{s.name}}]({{s.url}})
::: /each
```

The first block renders a heading and one list, or nothing when there are no breaking changes. The `:: #before` line is at the container's own level, not indented under the list item, which would put it inside the item. The second block renders one paragraph, such as "Rust, wgpu and WebAssembly".

Repeating a component. `each` around a component repeats it once per record, in order; the component knows nothing about the iteration:

```
::: each {in=(md.data.team) as=p}
::: card {title="{{p.name}}"}
{{p.role}}, since {{p.since}}
```

```
::: /card
::: /each
```

When a component must decide where each record's template goes (grouping, a calendar), it declares a template slot instead (section 7.5). `each`'s own signature is one: `in: array`, `as: name`, `slots: {default: {each: in, as: as}, before: {}, empty: {}}`; its `join` modes are its own.

Tables. A GFM table is one block, so `each` cannot produce table rows. Data tables come from a component (for example a pack's `:: table {rows=(md.doc.plans) cols=(["name", "price"])}`).

8.4 Computed text

`{{ }}` takes paths only. Computed or formatted text in prose uses the `@compmd/core` inline component `value`:

Param	Type	Meaning
`of`	`string`	required; rendered as text. Numbers and dates go through `string()` or a formatting function: `of=(string(size(rows)))`

```
The annual plan is :value[$120/year]{of=(md.currency(md.doc.price * 10,
"USD") + "/year")},
saving :value[$24]{of=(md.currency(md.doc.price * 2, "USD"))}.

# :value[Release notes]{of=(md.doc.strings[md.lang].title)}
```

The fallback is what plain renderers and failing processors show, so authors **SHOULD** write the value it normally produces.

8.5 Evaluation

- **Rendering is a function of its inputs only:** the document, its imports, its data and the render context. The same inputs **MUST** give the same output.
- **The render context** is the render time (`md.now()`, `md.today()`), the render's time zone, and the values of the document's inputs (section 8.7). The time zone (`md.tz`) is the document's `tz` key (section 2.1) when present, otherwise the processor's.
- **When to render is the processor's choice:** once at build time, on a schedule, when data or an input changes, or each time a reader opens the document. The spec sets no

refresh rule. A processor that renders again because something changed **MUST** produce what a fresh render with the new values would.

- There is no other reader context: no viewer identity, device or locale detection. Behaviour beyond inputs (tabs, toggles, platform detection) belongs in a host-language component.
- Evaluation is side-effect free and always terminates.
- **Output size.** Every processor **MUST** render documents whose expanded tree (section 12.3) has up to 100,000 nodes, and **SHOULD** render them within one second. Beyond 100,000 nodes, a processor **MAY** stop with F034 at the `each`, template slot or invocation whose expansion crossed the limit; processors **MAY** allow more. Nested `each` blocks are how documents get there: four nested 100-element lists are 100 million nodes.
- `compmd` definitions, including expressions and control flow, still need no sandbox (section 15).

8.6 Data imports

The `data` key under `md` loads data from files or URLs into `md.data`, for use in expressions and interpolation.

```
---
md:
  version: 1
  data:
    - from: ./team.yaml
      as: team
    - from: ./plans.csv
      as: plans
      schema: {name: string, price: number, seats: number}
    - from: "@myorg/ui/icons.json"
      as: icons
    - from: ./site.comp.md
      as: site
    - from: https://api.example.com/status.json
      as: status
      cache: 1h
      fallback: ./status.json
---

::: each {in=(md.data.team.filter(p, p.active)) as=p join=list}
- **{{p.name}}**, {{p.role}}
::: /each
```

Key	Type	Rule
`from`	string	required; a relative path, a file inside an imported pack (`@scope/pack/file`), an `https://` URL, or a `host:` source (below); **MAY** interpolate inputs (below)
`as`	name	required; the name the loaded value gets in the document: `as: team` makes it `md.data.team`
`schema`	declarations (section 7.3)	optional; for a list of records, the type of each field, and `from` to rename a key
`cache`	duration (`30s`, `10m`, `1h`, `1d`)	optional; remote only; how long a fetched copy may be reused
`fallback`	local path	optional; remote and `host:` only; loaded when the source fails
`delimiter`	one character	optional; CSV only; the field separator, default `,` (a tab for `.tsv`)

Formats are chosen by extension, or by `Content-Type` for URLs:

- `.yaml`, `.yml` and `.json` load as their natural value: YAML with the core schema (section 2.1), JSON numbers as `double`s.
- `.csv` and `.tsv` load as a list of maps keyed by the header row (below).
- `.comp.md` contributes its author's frontmatter keys only (what is `md.doc` in that document).

CSV follows RFC 4180: fields separated by the `delimiter`, `"` quotes a field and `""` writes a quote, quoted fields may hold separators and line breaks, lines end in LF or CRLF, the text is UTF-8 (a byte order mark is ignored). The first row is the header and is required.

- Without `schema`, every value is a string, and an empty cell is `""`.
- With `schema`, cells are read by the declared type: numbers with `.` as the decimal mark, booleans as `true` or `false`, dates as ISO dates (section 8.2). An empty cell is `null` for an optional field (`number?`) and F027 otherwise. Decimal commas (`1,50`) are not numbers; declare such columns as `string`.
- A row with more or fewer cells than the header is F027 for that record. A header with a duplicate or empty name is F026.

Typing. With `schema`, expressions over the data are type-checked like param expressions, and each record is validated when loaded (F027). Without it, the data is `dyn` and type errors surface at evaluation (F024); for data without a `schema`, the F024 message **SHOULD** suggest adding one.

Renaming. A `schema` entry's `from` names the key in the data, so keys that aren't identifiers (spreadsheet headers with spaces or `-`) get names paths can use:

```
md:
  data:
    - from: ./sales.csv
      as: sales
      schema:
        product: {from: "Product", type: string}
        unit_price: {from: "Unit price", type: number}
        units_sold: {from: "units-sold", type: number}
```

```
::: each {in=(md.data.sales) as=s join=list}
- {{s.product}}: {{s.units_sold}} sold at {{s.unit_price}}
::: /each
```

Scope is lexical: data imported by a pack is visible to that pack's definitions only, never to documents importing the pack.

Relative paths resolve against the file they are written in, as `url()` does in CSS. This applies to data `from` and `fallback` values, imports (section 6.1), and link and image destinations in `compmd` definition bodies: `![](./icons/star.svg)` in a pack's `card` is the pack's file wherever `card` is used. Slot content is written by the caller, so its paths resolve against the caller's document. A relative path cannot leave its root, the pack's folder or the document's project folder (which the processor decides): `../` beyond it is F026. Making a pack's files reachable in the output is the processor's job.

Values in data are never paths: a frontmatter value `logo: ./logo.png`, imported from another folder and used as `![]({{md.data.site.logo}})`, resolves where it is used. Data meant to be used elsewhere **SHOULD** hold absolute or root-relative URLs.

Host sources. A `from` that starts with `host:` names data the processor provides, such as a host application's own records; the processor defines what follows the colon. It is the one extension point for data from hosts:

```
md:
  data:
    - from: host:tasks?status=open&project={{md.input.project}}
      as: tasks
      schema: {title: string, due: date, project: string}
```

- Host data lands in `md.data` like any other import, with the same `schema`, F026, F027 and `fallback` rules. Hosts add no variables: expressions see only what section 8.2 lists.
- A `host:` source **SHOULD** have a `schema`, since the document can't otherwise know the data's shape, and two hosts may give the same name different shapes.
- A processor that doesn't provide the source reports F026 and uses `fallback`, or `null` (an `each` over it renders its `empty` slot). A processor **MAY** report W113 for each `host:` source, for portable documents.
- Access is the host's: the document carries no credentials. When the host's data changes, the processor renders again (section 8.5). Changing host data is not a data import; it belongs in a host-language component.

Values in `from`. A `from` string **MAY** interpolate inputs and frontmatter, so the reader's choices pick what is loaded:

```
md:
  data:
    - from: https://api.example.com/sales?from={{md.input.from}}&region=
{{md.input.region}}
      as: sales
    - from: ./sales-{{md.input.year}}.csv
      as: year
```

- Only `md.input` and `md.doc` paths may appear, never `md.data`, so data from one source can't be sent to another. Anything else is F031.
- An input that a host-language component can change (one bound to an `input<…>` param, section 7.4) may not appear either (F031): component code could otherwise write data into it and send it out (section 15). Inputs changed only through `@compmd/core`'s input components are the reader's own choices.
- Interpolation only (`{{ }}`), no expressions. A value is written as text by its type's rule.
- Each value is percent-encoded as one path segment or one query value: it cannot add `/`, `..`, `?`, `&` or `#`, so a local path stays in its folder and a URL gains no parameters.
- In a URL, the scheme and host are literal (F031 otherwise), so a processor checks the origin (F028) before rendering.
- `fallback` is always a literal path.
- When an input in a `from` changes, the processor loads again and renders again (section 8.5). Cached copies are keyed by the final URL or path.

Remote data.

- Only `https://` URLs are allowed. The scheme and host **MUST** be literal; values from inputs go only into the path and query (see "Values in `from`" above).

- The origin learns the values the reader picked, since they are part of the request.
- A processor fetches only from origins allowed by its user or host configuration; the document cannot grant access. A disallowed origin is F028.
- A processor that fetches while a reader views the document **MUST** ask the reader the first time a document needs an origin the reader hasn't decided on, and remember the answer. Origins the host configured need no question. Until the reader allows the origin, it is disallowed (F028), and the document renders with the `fallback`.
- Documents carry no credentials. Authenticated sources are configured per origin by the host, or reached through a host-language component.
- A failed fetch uses `fallback` if present; otherwise it is F026 and the data is `null` (an `each` over it renders its `empty` slot), in either mode: strict mode fails processing on it, as on any `F` error (section 10.1).
- A source the host is still getting is pending (section 6.4): it loads as a failed fetch does, its `fallback` or `null`, but without F026. The document renders again when it arrives, and when the host gives up on it, it is a failed fetch.

Errors. A missing, unreachable or unparseable source is F026, reported at the line and column inside the data file when available. A record that violates `schema` is F027, naming the record and field.

8.7 Inputs

An input is a typed value the reader can change, such as the start date of a dashboard. Inputs are declared in the `inputs` key under `md`, read as `md.input.<name>`, and changed through input components.

```
---
md:
  version: 1
  import:
    - from: "@myorg/charts@1"
      names: [chart]
  inputs:
    from: date = (md.today())
    region: all | eu | us = all
  data:
    - from: ./sales.csv
      as: sales
      schema: {date: date, region: string, amount: number}
---

:: date-input[From: today] {bind=from}
```

```
:: select-input[Region: all] {bind=region}

:: chart[Sales by day] {rows=(md.data.sales.filter(s, s.date >=
md.input.from && (md.input.region == "all" || s.region ==
md.input.region)))}
```

Declaring. Each entry under `inputs` names an input, a CEL identifier, and gives its type and default as a declaration (section 7.3). Every input **MUST** have a default, so a document always renders. Inputs belong to the document being rendered: `md.input` is available in document bodies and in `from` values (section 8.6), not in `compmd` definition bodies, which receive input values as params. An imported document's inputs are not used.

Values. An input's value comes from whoever runs the processor (a command-line argument, a URL query, a host) when given, otherwise from its default. A given value that doesn't match the input's type is rejected, and the default is used. Where values are kept between renders is the processor's business.

Input components. `@compmd/core` provides leaves, implemented by the processor (section 9), that show an input's value and let the reader change it:

Component	Edits inputs of type
`date-input`	`date`
`number-input`	`number`
`text-input`	`string`
`select-input`	an enum; its members are the choices (`input<enum>`)
`toggle-input`	`boolean`

- Each takes `bind`, a param of type `input<…>` naming a declared input (section 7.4). An undeclared name, or an input of a type the component cannot edit, is F030.
- The fallback is the input's label as plain renderers show it; authors **SHOULD** write the default value into it ("From: today").
- A processor that is not interactive renders an input component as a static display of the value, or as its fallback.
- When the reader changes a value, the processor renders again (section 8.5).

One direction. Values flow from inputs to expressions to output. There are no events, scripts or other state; `compmd` definitions stay free of code and need no sandbox (section 15). Host-language components change an input only through a param of type `input<…>` that the document binds to it (section 7.4).

9. Built-ins: the @compmd/core pack

Built-ins are an ordinary pack, `@compmd/core`, imported into every document. It declares one primitive the processor implements, `box`, and builds `stack`, `row` and `grid` on it in the CompMD language.

Component	Defined as	Lays out children	Own params
`box`	processor	per `display`	`display` (`block`, `flex`, `grid`; `contents` on a definition fence, section 9.2), `flex-direction`, `gap`, `align-items`, `justify-content`, `flex-wrap`, `grid-template-columns`, `grid-template-rows`, `grid-template-areas`
`stack`	CompMD, on `box`	top to bottom	`gap`, `align-items`
`row`	CompMD, on `box`	side by side, wrapping when there isn't room	`gap`, `align-items`, `justify-content`, `flex-wrap` (`wrap` by default, or `nowrap`)
`grid`	CompMD, on `box`	as cards, as many per row as fit (`min`), or on an exact grid	`min`, `gap`, `align-items`, `grid-template-columns`, `grid-template-rows`, `grid-template-areas`

- **Names.** A param that sets one CSS property is named after that property and takes its values (section 9.4), so people and tools can guess it. CompMD's own names are only for what no single CSS property does: `min`, `scroll`, `fill`, `sticky`, `show` and `hide`. `justify-content` also accepts `between` and `around` for `space-between` and `space-around`.
- **`grid`** with `min` sets `grid-template-columns` to `repeat(auto-fill, minmax(<min>, 1fr))`: `::: grid {min=14rem gap=m}` shows three cards across a wide page and one per row on a phone. With neither `min` nor a template, a grid has one column, so `::: grid {md:grid-template-columns="3fr 2fr"}` puts two blocks side by side from `md` up and stacks them below it.
- **Exact grids** place their children with `grid-area` (section 9.2): `::: grid {grid-template-columns="12rem 1fr" grid-template-areas="'head head' 'side main'"}`.

`@compmd/core` also exports `if` and `each` (section 8.3), `value` (section 8.4) and the input components (section 8.7), which the processor implements too.

Components the processor implements. In a pack a processor provides (`@compmd/core`, an ambient pack, section 6.3), a signature fence with no definition and no import means the processor implements that component: the pack states its signature, so documents get validation and fallbacks for it like for any component. In a document, a signature fence with nothing to attach to is F002. A processor targets a new output (PDF, terminal) by implementing `box` alone.

````
---
md:
  version: 1
  export:
    pack: "@compmd/core"
    version: "1.0.0"
    names: "*"
---

```compmd component=box signature
params:
  display: block | flex | grid | contents = block
  flex-direction: row | column = row
  gap: space = none
  align-items: start | center | end | stretch = stretch
  justify-content: start | center | end | between | around | space-between
| space-around = start
  flex-wrap: wrap | nowrap = nowrap
  grid-template-columns: tracks?
  grid-template-rows: tracks?
  grid-template-areas: areas?
slots: [default]
```

```compmd component=row {display=flex flex-direction=row gap=(gap) align-
items=(align_items) justify-content=(justify_content) flex-wrap=
(flex_wrap)}
params:
  gap: space = none
  align-items: start | center | end | stretch = stretch
  justify-content: start | center | end | between | around | space-between
| space-around = start
  flex-wrap: wrap | nowrap = wrap
slots: [default]
---
````
````

````
{{md.slot}}
```
````

`stack` and `grid` follow the same pattern: the fence styles the component's box, and the body is the slot. Because `@compmd/core` is written in CompMD, third-party packs can define their own layouts the same way.

Positioned layout. Any block can be placed at a position in its box with the `inset-*` common params (section 9.2), like a note pinned on a board:

```
::: box {height=12rem padding=m}
This week's notes, pinned where they matter.

::: box {inset-inline-start=60% inset-block-start=3rem width=30%}
Skipped Tuesday: travel.
::: /box

:: badge[3 new] {inset-inline-end=8px inset-block-start=8px}
::: /box
```

The paragraph flows as usual; the note and the badge are placed over it.

9.1 Items

Each top-level block in a layout container's default slot is one **item**: a paragraph, list, table, heading, leaf, or nested container. To make several blocks one item, wrap them in `::: box`. A `compmd` component is one item, its box, unless its definition is `display=contents`; then its blocks are the container's items (section 9.2).

9.2 Common params

Any container or leaf (built-in or not) accepts these params, applied to its outer box:

Param	Values	Example
`width`, `height`, `min-width`, `max-width`	a length, a percentage or `auto`	`width=40%`
`padding`	one or two spaces (`none`, `s`, `m`, `l` or a length): block, then inline	`padding=m`, `padding="s l"`
```

Param	Values	Example
`padding-inline`, `padding-block`	one or two spaces (`none`, `s`, `m`, `l` or a length): start, then end	`padding-inline=l`
`padding-inline-start`, `padding-inline-end`, `padding-block-start`, `padding-block-end`	a space (`none`, `s`, `m`, `l` or a length)	`padding-block-start=s`
`border`	a token (`none`, `subtle`, `strong`) or a CSS border	`border=subtle`, `border="1px solid"`
`border-radius`	a token (`none`, `s`, `m`, `l`, `full`) or a length	`border-radius=m`
`background`	a token (`none`, `subtle`, `strong`, `accent`) or a CSS color	`background=subtle`
`grid-area`	an area name of the parent's `grid-template-areas`	`grid-area=side`
`order`	`first`, `last`, `none`	`order=first`
`align-self`	`start`, `center`, `end`, `stretch`	`align-self=center`
`justify-self`	`start`, `center`, `end`	`justify-self=end`
`aspect-ratio`	width over height, or `none`	`aspect-ratio=16/9`
`inset-inline-start`, `inset-inline-end`, `inset-block-start`, `inset-block-end`	a length, a percentage or `auto` (the default)	`inset-inline-start=16px`
`scroll`	a flag, or `true` or `false`	`scroll`
`fill`	a flag, or `true` or `false`	`fill`
`sticky`	`top`, `bottom`, `none`	`sticky=top`
`show`, `hide`	`sm`, `md`, `lg`	`show=md`

- **The outer box.** Every component invocation that is a block has one box, as a custom element does. A host-language component's box is its `target` (section 7.4). A `compmd` component's box holds its body, whatever its params, body or data: `::: row {height=8rem`

`.toolbar}` is one box with the row's layout, the height and the class, so a block placed by position in it uses that height and `.toolbar > *` selects the row's items.

  - **The definition styles its box** with its fence's params (section 7), and **the caller's params override them** as a later cascade layer does in CSS: per property, a shorthand counting as its longhands (`padding` covers `padding-block`, `padding-inline` and the four sides), and from the caller's breakpoint up (breakpoints decide where a value applies, not which wins). Classes add up. A `card` fenced `{padding=s md:padding=m}` called with `{padding=l}` has `l` at every width; called with `{md:padding=xl}` it has `s` below `md` and `xl` from `md` up. A `fill`, `grid-area` or `sticky` the box needs wherever it's used goes on the fence: a definition whose body is a filling list view is fenced `{fill}`, or its box is as tall as the list's content.
  - **`display=contents`** on a `compmd` fence means the component has no box: its blocks are its parent's items (section 9.1), and blocks it places by position are placed in its parent's box. It is written as a bare value with no breakpoint, and only on a definition fence (F009 elsewhere). The fence then takes no other param, and an invocation gives such a component no common params, `class` or `id`, since there is no box for them (W120 for each).
  - Insets follow CSS: a caller's `inset-inline-start` doesn't remove the fence's `inset-inline-end`, so the block stretches between them; `inset-inline-end=auto` moves it instead.

- **Sizes include padding.** `width`, `max-width` and `aspect-ratio` measure the block with its padding.
- **Output order is source order.** No layout param (`order`, `grid-area`, `grid-template-areas`, `inset-*`, breakpoints) changes the order of the output: screen readers, tab order, copy and paste, and plain renderers follow the source, so content is written in reading order. `order`, `grid-area` and `inset-*` move blocks on screen only. Authors **SHOULD** keep the visual order close to the source order where blocks hold links or inputs, so keyboard focus doesn't jump around the screen.
- **`justify-self`** is CSS's in a grid. In a parent that lays out with flexbox (`stack`, `row`), `end` pushes the block, and the blocks after it, to the end.
- **`scroll`** makes the block's content scroll inside it, so wide content (a table on a phone) doesn't scroll the page.
- **`fill`** makes a block take the height its parent leaves; at the top level, the height of the area the processor renders into. Nested `fill` and `scroll` blocks scroll instead of growing past their parent. In a document that flows, with no fixed height, `fill` does nothing.
- **`sticky`** keeps the block at that edge of the nearest area that scrolls it.
- **A block with a position** (an `inset-*` param that isn't `auto`) is taken out of the flow and placed in its **nearest box** in the expanded tree (section 12.3): a `box` (a `compmd` component's box among them), a host-language component's outer box, or one item of a host-language component's slot. `each`, `if`, slots of `compmd` components and

`display=contents` components don't count, and neither do list items, quotes and table cells. At the top level, it is placed in the area the processor renders into.
    - It is placed against that box's padding edge: `inset-inline-start` from the start edge (the left in left-to-right languages, the right in right-to-left ones), `inset-inline-end` from the end edge, `inset-block-start` from the top and `inset-block-end` from the bottom. Percentages are of the box's width or height.
    - `width` and `height` size it. Without a `width` it fits its content; with both inline insets it stretches between them.
    - The other blocks flow as if it weren't there. It is drawn over them; among positioned blocks, later ones are on top. Reading order stays source order.
    - `order`, `sticky`, `fill`, `grid-area`, `align-self` and `justify-self` don't apply to it.
    - A box that places blocks needs a height, from `height`, `aspect-ratio` or `fill`; its children don't give it one (W119 otherwise).
- **`show=md`** shows a block from `md` up, **`hide=lg`** hides it from `lg` up, and both together give a range (section 9.3). A hidden block is still rendered, so its state survives a resize, but it is hidden from assistive technology too, not only from view.

9.3 Breakpoints

A layout or common param **MAY** carry a breakpoint prefix: `md:gap=l` overrides `gap` from the `md` breakpoint up. The breakpoints are fixed: `sm` is 480 px, `md` 768 px and `lg` 1024 px. Documents start from the narrowest layout and override upwards.

- **Breakpoints follow a block's nearest box, not the window.** Every param of a block follows the width of its nearest box (section 9.2): its layout (`md:gap`, `md:grid-template-columns`) as well as its size, padding, `show`, `hide`, `order`, `sticky` and self-alignment. At the top level, that is the area the processor renders into. A document shown in a narrow column lays out as on a phone. A processor whose output has no width that changes (a PDF page) uses the width it has. In HTML this is a CSS container query on the nearest box, since CSS lets an element's styles follow an ancestor's width, not its own.

- **A component follows the box it is in, not its own width.** A `card` fenced `{padding=s md:padding=m}`, in `::: grid {min=14rem}` on a page 1000 px wide, has `m` in every card: the grid is 1000 px wide, though each card is about 320 px.

- `grid-area`, `scroll`, `fill`, `show` and `hide` take no breakpoint. Positions do: `md:inset-inline-start=60%` places a block from `md` up and leaves it in the flow below, and `md:inset-inline-start=auto` does the reverse. W119 is checked at each breakpoint. A breakpoint on a param that takes none (`md:show=md`, or a param of a host-language component), or a breakpoint other than `sm`, `md` and `lg`, is F008, and the param is ignored.

- **A box that blocks follow has no width from its content.** A box whose blocks use breakpoints (a breakpoint prefix on any of their params, or `show` or `hide`) is the box they follow, so where its width would come from its content, it shrinks to nothing. Such a box needs a `width` when it is:
  - in a `row` (any box with `display=flex` and `flex-direction=row`);
  - in a `stack` (any box with `display=flex` and `flex-direction=column`) and not stretched across it: its `align-self`, or else the stack's `align-items`, isn't `stretch`, or its `justify-self` is `center` or `end`;
  - in a `grid` (any box with `display=grid`), with a `justify-self`;
  - positioned (section 9.2), without both `inset-inline-start` and `inset-inline-end`.

  Without a `width` it is W114, checked at each breakpoint. In a box with `display=block`, self-alignment doesn't change a block's width. A block whose own params use breakpoints follows its nearest box and needs nothing.
- A breakpoint value given to a param of a component defined in CompMD applies wherever the definition passes that param on: `md:gap=l` on a `row` reaches its `box`. When an expression in the definition's body uses such a param, the processor evaluates it once per breakpoint, mobile-first (at `md`, the `md` value, or else the `sm` one, or else the base), and each result that differs from the base value becomes a param with that breakpoint: `::: grid {min=14rem md:min=20rem}` gives `grid-template-columns` and `md:grid-template-columns`.

9.4 Values and spacing

- **Visual tokens** (`background`, `border` and `border-radius`, types `surface`, `border` and `radius`) work like spaces: a token or a CSS value. The tokens are CSS custom properties with defaults built from the text color, so they suit light and dark text with no stylesheet, and a theme may change them:

Token	Custom property	Default
`background=subtle`, `strong`	`--compmd-surface-subtle`, `--compmd-surface-strong`	`color-mix(in srgb, currentColor 6%, transparent)`, 12%
`background=accent`	`--compmd-surface-accent`	`color-mix(in srgb, var(--compmd-accent, #0969da) 12%, transparent)`
`border=subtle`, `strong`	`--compmd-border-subtle`, `--compmd-border-strong`	`1px solid color-mix(in srgb, currentColor 20%, transparent)`, 45%

Token	Custom property	Default
`border-radius=s`, `m`, `l`, `full`	`--compmd-radius-s`, `-m`, `-l`, `-full`	`4px`, `8px`, `16px`, `9999px`

`none` means no background, border or rounding. No token is a CSS keyword of the same param, so a value is never ambiguous.

- **Spaces** (`gap` and the padding params, type `space`) are a token, `none`, `s`, `m` or `l`, or a length. The tokens are the CSS custom properties `--compmd-space-s`, `--compmd-space-m` and `--compmd-space-l`, by default `0.5rem`, `1rem` and `2rem`, which a processor or stylesheet may change. Documents **SHOULD** use tokens; lengths are for layouts that need exact sizes.
- **Lengths** are `px`, `rem`, `em` and `%`. **Tracks** (`grid-template-columns`, `grid-template-rows`, type `tracks`) also take `fr`, `auto`, `min-content`, `max-content`, `minmax()` and `repeat()`. **Areas** (`grid-template-areas`, type `areas`) are quoted rows of area names or `.`, all the same length (`"'head head' 'side main'"`). **Colors** and **borders** take CSS syntax.
- **The vocabulary is closed.** The params in this section are all the layout and styling a document can write. An unknown param is F008 and a value outside its grammar is F009, so no param can pass raw CSS through.
- **Directions are logical.** Every param says `start` and `end`, `inline` and `block`, never left, right, top or bottom, so a document laid out for a left-to-right language reads correctly right to left.
- **Meaning.** Each param means what the CSS property it names means (section 13.1). An HTML processor emits that CSS. A processor for another output (PDF, terminal) implements the same meaning, as far as its output allows.

9.5 Which box breakpoints follow (informative)

Each example says which box a block's breakpoints follow (section 9.3), on a page 1000 px wide.

- **At the top level.** `::: grid {md:grid-template-columns="1fr 1fr"}` follows the page: two columns.
- **In a narrow box.** The same grid inside `::: stack {width=50%}` follows the stack, 500 px wide: one column. The stack is the box the grid follows.
- **Only its own params.** `::: box {md:padding=l}` at the top level follows the page. It isn't a box anything follows, since no block in it uses breakpoints.
- **A card in a card grid.** Each card follows the grid, not its own width (section 9.3).

- **In a list item, a quote or an alert.** `:: chart[Sales] {md:height=20rem}` in a list item inside a `box` follows the box: list items and quotes aren't boxes.
- **In a `display=contents` component.** It has no box, so its blocks follow the caller's box.
- **In a host-language component's slot.** A grid in a `tabs` component's slot follows its item, whose width the component gives (section 7.4).
- **In `each` and `if`.** They expand to their content, so its blocks follow the box around the `each` or `if`.

---

10. Errors, diagnostics and recovery

Every diagnostic **MUST** name its code, the line and column where it was detected, the construct involved, and any related location (such as the opener of an unclosed container). One mistake **MUST** produce one primary diagnostic, not a cascade.

10.1 Modes

- **Strict mode:** any `F` error fails processing. Intended for CI and publishing.
- **Lenient mode:** errors are reported, recovery rules apply, and processing continues with fallbacks. Intended for previews and editors.

10.2 Catalog

Code	Condition	Reported at
F001	Unsupported major version (`md.version`)	line 1
F002	Unknown component name (diagnostic **SHOULD** suggest the nearest declared name)	the name
F003	Closer names no open container while one is open (a misspelt name)	the closer, naming the innermost open container and its line
F004	Container not closed before an enclosing closer or end of file	the opener, and where it was auto-closed
F005	Closer while no container is open	the closer
F006	Undeclared slot name	the slot line

Code	Condition	Reported at
F007	Named slot given twice in one container	the second slot line, naming the first
F008	Unknown param, or a breakpoint the param doesn't take	the param
F009	Param value does not match its type or CSS grammar, or a code renderer's signature declares `code` as anything but a required `string` (section 7.3)	the value, or the declaration
F010	Required param missing	the opener or leaf
F011	Malformed param list (unterminated string, unbalanced bracket, invalid JSON, a character a bare value can't hold, an empty value)	the `"`, `(`, `[` or `{` that doesn't close, the character a bare value can't hold, or the `=` with no value
F012	Continuation line not following an opener, leaf, or continuation	the line
F013	Duplicate local name from definitions or explicit imports, or a keyword (`if`, `each`, `value`, `box`) defined, imported, exported or used as `as`; the name `md` declared; a CEL reserved word or type name as a `compmd` definition's param, an `as` name or an input (section 3.1); two params, or two slots, of one component differing only in `-` and `_`	the second declaration
F014	Unknown interpolation path	the path
F015	Slot line outside a container	the line
F016	Malformed component line (starts with `::` but matches no form)	the line, with the expected form
F017	Param given twice, or `id` together with `#name`, or a code renderer's `code` param given on a code block's info string (section 7.7)	the second occurrence, or the param
F018	Body given to a component that declares no slots	the opener
F019	Recursive component definition	the invocation, or the rendered code block's language, that closes the cycle

Code	Condition	Reported at
F020	Import cycle	the `from` that closes the cycle
F021	Imported name not exported by its source	the `names` entry
F022	`export.pack` or `export.version` disagrees with the registry manifest	the `export` key
F023	Name provided by two wildcard imports, or by a wildcard import and `@compmd/core`, is used; or a code block's language rendered by two code renderers at the same level (section 6.3)	the use, naming both sources
F024	Expression parse, type or evaluation error, or a list or a map without a `name` interpolated into text	the column inside the expression, or the interpolation
F025	each iteration does not match the shape its join mode requires	the each opener, naming the first offending line
F026	Data or import source missing, unreachable, unparseable or not provided (`host:`), a relative path leaving its root, a bad CSV header, or a pack the processor's configuration doesn't allow or that exceeds its size cap (section 6.4); never for a source the host says is pending (sections 6.4, 8.6)	the data entry or the import's `from`, and the position inside the data file when available
F027	Data record violates its schema	the record and field
F028	Remote data origin not allowed by processor configuration	the from value
F029	Exported host-language component has no signature fence	the export.names entry
F030	A param of type `input<…>` (an input component's `bind`, a host component's bound param) names an undeclared input, or an input of another type	the param's value
F031	A `from` interpolates something other than `md.input` or `md.doc`, an input a host-language component can change, or a value in a URL's scheme or host	the interpolation
F032	Closer names a container opened outside its list item or block quote	the closer, naming the opener's line

Code	Condition	Reported at
F033	Frontmatter isn't YAML 1.2 core, uses tags, anchors or aliases, repeats a key, or exceeds the processor's size cap	the frontmatter, at the offending position
F034	Expansion larger than the processor allows (at least 100,000 nodes, section 8.5)	the `each`, template slot or invocation that crossed the limit
F035	Pack does not match its import's `integrity` hash	the `integrity` key
F036	Param list on a signature fence; ignored	the `{`
F037	A declaration's type isn't in the type grammar (section 7.3), in a signature, an input or a data `schema`; the declaration takes any value (diagnostic **SHOULD** suggest the nearest type name)	the type name
W101	Leaf without fallback text (plain renderers show nothing, and a host component has no text alternative), or a code block a code renderer renders without `alt` (section 7.7)	the leaf, or the code block's opening fence
W102	Definition language not supported by this processor; fallback rendered	the invocation
W103	each default slot is a single list item and join is unset	the each opener
W104	Signature fence and component disagree (missing prop, type or slot)	the signature entry, naming the component's declaration
W105	Local host-language component has params that could not be typed (dyn)	the definition fence
W106	`{{` followed by something that isn't a path; kept as text	the `{{`
W107	Inline component followed by a space and a param list, which is read as text	the `{`
W108	Container opener or leaf directly after a paragraph line, without a blank line	the component line

Code	Condition	Reported at
W109	Interpolated link or image URL with a scheme other than `https:`, `http:` or `mailto:`; rendered without the link	the URL
W110	Document declares a later minor version than the processor implements	`md.version`
W111	Unknown key under `md`; ignored	the key, naming the nearest known key
W112	Local definition or explicit import shadows a `@compmd/core` component	the definition or `names` entry
W113	Name from an ambient pack, a `host:` data source, or a library the processor offers, used (optional; for portable documents)	the use, the data entry, or the component's definition
W114	Box whose blocks use breakpoints, without a `width` where its width would come from its content (section 9.3)	the box
W115	Host metadata given for a document that has its own frontmatter; the host's metadata is ignored	line 1
W116	Line starting with `::` inside an HTML block, which makes it HTML, not CompMD	the line
W117	Exported host-language component without `runtime`	the definition fence
W118	Host-language component threw while mounting or updating, or failed (section 7.4); fallback rendered	the invocation
W119	A box that places blocks by position has no `height`, `aspect-ratio` or `fill` at some width (section 9.2)	the box's opener; at the top level, the placed block
W120	A param for a box given to a `display=contents` component, which has none, on its fence or by an invocation; ignored (section 9.2)	the param

Code	Condition	Reported at
W121	Host-language component's code can't load: an import of something it may not import (section 7.6), a compile error, a built module that doesn't match its file, or a runtime or library its pack's `peerDependencies` name that the processor doesn't have in that range (section 6.4); fallback rendered	the invocation, or the import's `from` for `peerDependencies`

10.3 Recovery rules (lenient mode)

Recovery follows four principles:

1. **Keep the structure** whenever a component's name is readable.
2. **Drop the smallest broken piece** (a param, a line, a record), never author content.
3. **No cascades:** a diagnostic caused by one already reported is not reported.
4. **Show content rather than hide it** when unsure.

Code	Recovery
F001, F033	No CompMD parsing: the file renders as CommonMark and GFM
F003	The closer closes the innermost open container
F004	A closer naming an outer open container closes every container inside it, one F004 each; at the end of the file, every open container is closed there
F005	The closer line is ignored
F032	The closer line is ignored; the container it names closes at the end of its own parent
F007	The second slot's content is appended to the first
F011	On a block line, the component keeps the params before the break and is marked `recovered`. In an inline component, the component keeps no params, and the rest of the line from `{` is text
F012	The continuation line is ignored
F015	The slot line is ignored; what follows is normal content
F016	The line is plain text
F017	The first occurrence wins

Code	Recovery
F002, F010, F019, F023, F030	The component renders its fallback (section 4.5)
F013	The first declaration wins
F020, F021, F035, F026 for an import	The import, or the name, is dropped; its uses render fallbacks without F002
F022	The document's `pack` and `version` are used
F029	The export's signature is derived, as for a local component
F006	The undeclared slot's content joins the default slot
F008	The param is ignored
F009, F024 in a param	The param is treated as absent: its default is used, or the component renders its fallback if the param is required
F014, F024 in an interpolation	The interpolation is shown as written
F018	The component renders without a body, and the body follows it as normal content
F024 in an evaluation	The component renders its fallback
F025	The iterations are appended, as with `join=none`
F026 for data, F028, F031	The data is its `fallback` file if given, otherwise `null` (an `each` over it renders its `empty` slot)
F027	The record is dropped
F034	What was expanded before the limit is kept; the component that crossed it renders its fallback (an `each` or template slot, its template once)

Warnings need no recovery: each describes what the processor does anyway.

10.4 Diagnostics in imported definitions

- A diagnostic points where the mistake is, in the defining document; the invocations that led there are `related` locations, innermost first.

- Errors found when a definition is loaded (parse and type errors) are reported once per definition, not once per use.
- Evaluation errors carry the invocation chain that triggered them. The same code at the same position is reported once.
- A document in a pack is named by the pack, its version and its path: `@myorg/ui@1.2.0/card.comp.md:7:24`.

10.5 Machine format

Processors that report diagnostics to other tools use this JSON form:

```
{
  "code": "F024",
  "severity": "error",
  "message": "no '+' for string and int",
  "file": "ui.comp.md",
  "range": { "start": { "line": 7, "column": 24 }, "end": { "line": 7,
"column": 32 } },
  "related": [
    { "file": "release.comp.md", "range": { "start": { "line": 12,
"column": 1 } }, "message": "invoked here (card)" }
  ],
  "stage": "validate"
}
```

- Lines and columns are 1-based; columns count Unicode code points. `end` is optional.
- `severity` is `error` for F codes and `warning` for W codes; `stage` is the stage that found it (section 11).
- `fix`, optional, is an edit that fixes the diagnostic, when a processor has one to suggest: the text at `range` (on one line, in the diagnostic's document) becomes `text`. A misspelt name or token gets one, such as `"fix": { "range": { "start": { "line": 3, "column": 17 }, "end": { "line": 3, "column": 22 } }, "text": "subtle" }` for `border=subtl`. An editor offers it as a quick fix.
- Messages are not normative. The form maps one to one onto an LSP `Diagnostic`, and a `fix` onto a code action; an LSP adapter converts positions to 0-based UTF-16.

11. Processing model

A processor works in stages. Each stage needs only what its row lists, and reports only the diagnostics listed for it.

Stage	Needs	Produces	Diagnostics
1. Parse	the document's text	the document tree (section 12)	F001, F003–F005, F007, F011, F012, F015–F017, F032, F033, F036, W101, W106–W108, W110, W111, W115, W116
2. Resolve	the tree, imported documents and packs	each name bound to a definition and signature	F002, F009 (a code renderer's `code`), F013, F019–F023, F026 (imports), F029, F035, W104, W105, W112, W113, W117, W121 (`peerDependencies`)
3. Validate	the resolved tree and signatures	a checked tree, params typed	F006, F008–F010, F011 and F017 (a rendered code block's info string), F014, F018, F023 (a code block's language), F024 (types), F028, F030, F031, W101 (a rendered code block), W102, W103, W120
4a. Load	the checked tree and the input values	the loaded data	F026, F027, W113 (`host:`)
4b. Evaluate	the checked tree, the loaded data, the render context	the expanded tree (section 12.3)	F024 (evaluation), F025, F034, W109, W114, W119
5. Render	the expanded tree	output (HTML, PDF, …)	W113 (offered libraries), W118, W121

- **Parsing needs no imports, signatures or data,** so editors, linters and formatters can parse any document. It reads the frontmatter as YAML, but interprets only `md.version` and `md.syntax` (and splits `from` strings into parts, section 12.2). Params stay as written and untyped: `count=3` is `"3"` until validation.
- **Resolving MAY load packs** (from a registry, a cache or the file system), or take those its host is still getting as pending (section 6.4), as loading takes pending data (section 8.6).
- **Stages 1–3 depend only on the document and its imports.** A processor **MAY** keep their results until either changes, and run only stages 4a–5 when data or an input changes.

- **Stage 4b is a function of its inputs** (section 8.5): the same checked tree, loaded data and render context always give the same expanded tree. All input and output happens in 2 and 4a.

---

12. Document tree

The tree is the result of stage 1 and the subject of conformance tests (section 13). CommonMark and GFM content uses mdast nodes, and the frontmatter is mdast's `yaml` node. CompMD adds the nodes below. Every node carries its source position (unist `position`), which diagnostics refer to.

12.1 CompMD nodes

Node	Fields
`componentContainer`	`name`, `params`, `children` (the default slot), `slots` (one `componentSlot` per named slot, in source order), `closePosition`
`componentSlot`	`name`, `children`
`componentLeaf`	`name`, `fallback` (inline children), `params`
`componentInline`	`name`, `fallback` (inline children), `params`
`componentDefinition`	`name`, `language`, `signature` (null when the fence has none), `params` (the fence's param list, section 7); for `compmd`, `children` (the parsed body); for host languages, `source` (the text) and `runtime` (null when the fence has none)
`componentSignature`	`name`, `signature` (a signature fence, section 7.1)
`interpolation`	`path`, `block`

`params`, `fallback`, `children` and `slots` are always present, as empty lists when there is nothing. A leaf written without `[…]` has an empty `fallback`.

A param is `{key, breakpoint, kind, value, position}`:

- `kind` is `bare`, `string`, `json`, `expr`, `flag`, `class` or `id`; `key` is null for `class` and `id` (`.wide`, `#intro`); `breakpoint` is null unless the key has one (`md:gap`, breakpoint `md`, key `gap`).

- `value` is the text as written: for `expr`, without the outer parentheses; for `flag`, null. A `string`'s value is a list of text and `interpolation` parts, with escapes resolved.
- Params from continuation lines join their opener's or leaf's `params`; their positions keep the line they came from.
- A repeated param (F017) keeps only its first occurrence.

A signature is `{params, slots, headings}`, plus `renders`, the list of its languages, for a code renderer, whose `params` include `code` (section 7.7): `slots` is the list of slots, or null when the signature has no `slots` key, each a slot name or, for a template slot, `{name, each, as}`; `headings` is `"relative"` or `"absolute"` (section 7.2); each param is `{name, type, optional, default, doc}`:

- `type` is a type name (`"string"`, `"length"`, ...), `{"enum": [members]}`, or `{"input": type}` for `input<type>`.
- `default` is null, or a param value from the short form (`{"kind": "bare", "value": "tip"}`), or `{"kind": "yaml", "value": …}` from the long form.
- `doc` is null unless the long form gives one.

12.2 Rules

- **A block interpolation.** A `interpolation` alone in its paragraph replaces the paragraph and has `block: true`; it can hold block content such as a slot. Elsewhere it is inline (`block: false`).
- **Interpolation in URLs.** Links, images and link reference definitions keep mdast's `url` as written; when the URL holds an interpolation they add `urlParts`, the same list of text and interpolation parts as a string param.
- **Frontmatter.** The `yaml` node adds `parsed`, the frontmatter as parsed YAML, in which each data import's `from` is such a list of parts. (Not `data`: unist reserves that field for tools.) When CompMD processing doesn't apply (F001, F033), `parsed` is absent and the rest of the document is plain CommonMark and GFM.
- **Recovery.** A node made by a recovery rule (section 10.3) has `recovered: true`. A container closed by recovery has `autoClosed: true` and no `closePosition`. Both fields are absent otherwise.

12.3 The expanded tree

Stage 4b replaces every `compmd` definition, `if`, `each`, `value` and `interpolation` with what it produces. The expanded tree is content only: definition fences, signature fences and the `yaml` node are left out (the parse tree keeps them). It holds mdast nodes, plus `componentContainer`, `componentLeaf` and `componentInline` nodes only for components the

processor renders itself: `box`, the input components, and host-language components; and `componentPending` nodes for uses of components still on their way.

- **Params** of those nodes are `{key, breakpoint, value, position}`, with `value` evaluated (JSON): `{"key": "gap", "breakpoint": "md", "value": "l"}`. All classes, from `.name` and `class`, merge into one `class` param, in written order without duplicates; `#name` is the `id` param. A param whose value is null is left out. `position` is where the param is written in the document, and absent when a definition computed it.

- **A param bound to an input** (type `input<…>`, section 7.3) has the input's current value as its `value`, and `input`, the input's name: `{"key": "selected", "breakpoint": null, "value": "eu", "input": "region"}`. A renderer gives the component the value, and `set` changes the named input (section 7.4).

- **The root lists the document's inputs** in `inputs`, when it has any: each one's type, an enum's `members`, and the value the render used: `{"region": {"type": "enum", "members": ["all", "eu", "us"], "value": "eu"}, "since": {"type": "date", "value": "2026-10-15"}}`. Input components read their choices from it, so a renderer needs nothing but the expanded tree.

- **Slots** are expanded. A template slot (section 7.5) has `children: []` and `instances`, one `{value, children}` per element, with the slot content expanded for that element.

- **A block `compmd` component** expands to a `box` holding its expanded body. The box's params are the fence's, evaluated, with the caller's common params, `class` and `id` applied over them: a fence param that a caller's param covers (the same property, or a part of it), at the caller's breakpoint or a wider one, is left out, and a fence shorthand the caller covers only in part is written as its longhands first. A `display=contents` component expands to its body alone. A host-language component's params hold its fence's params the same way.

- **Headings** from `compmd` definition bodies have their computed `depth` (section 7.2).

- **A use of a pending component** (section 4.5) is a `componentPending` node: `name`, the component's local name, `pack`, the pending pack as the import names it (`@kit/mermaid`), and `children`, the fallback the use renders (section 4.5), expanded. For a leaf or an inline component that is its fallback text, for a container its slots as Markdown, and for a code block a code renderer renders the code block, without its `component`. It is inline in inline content, and a block elsewhere. A renderer that doesn't know the node renders its children.

- **A code block a code renderer renders** (section 7.7) stays a `code` node, with its `lang` and `value`, and adds `component`, `{name, params}`, with params as a component node's, its `code` param holding the block's text, so a renderer passes them on as for any leaf; and `alt`, when given: `{"type": "code", "lang": "mermaid", "value": "graph LR; A --> B",`

`"component": {"name": "mermaid", "params": [{"key": "code", "breakpoint": null, "value": "graph LR; A --> B"}]}, "alt": "A, then B"}`. A renderer runs the component, and shows the code block as its fallback.

- **Origins** (the `origins` feature, section 13.1). Every expanded node except the root has `origin`, the range of the processed document that made it: `start` and `end` points as in `position`, and `occurrence`, how many nodes before it in the same list of children have the same range (a template slot's instances count as one list). Lines and columns are 1-based, offsets 0-based, and columns and offsets count code points (section 10.5). The range is:
  - for a node from a `compmd` definition's body, local or imported, at any depth: the origin of the invocation in this document that led to it;
  - for text that a `value` or an interpolation produced: its parent's origin;
  - for a node made by recovery (section 10.3): the text it stands for, such as the component line whose fallback it is;
  - for any other node: where it is written. A container's range runs from its opener to the end of its closer, or to where recovery closed it.

  The iterations of an `each` and the instances of a template slot share their template's range and differ by `occurrence`. Origins let an editor keep what it rendered for a node through an edit: it follows the last render's origins through the edit and matches the new nodes by them. How it does that is not part of this spec.

12.4 Serialization

Trees are serialized as JSON with the field names above. The JSON schema, `conformance/compmd-tree.schema.json` (JSON Schema 2020-12), defines both trees: `#/$defs/parseRoot` for the parse tree (sections 12.1, 12.2) and `#/$defs/expandedRoot` for the expanded tree (section 12.3). It checks this spec's nodes and params strictly and other nodes as mdast nodes; positions are optional. Every expected tree in the conformance set fits it.

---

13. Conformance

13.1 What conforms

- **A conforming document** is one a processor in strict mode reports no F diagnostic for.
- **A conforming parser** implements stage 1: for every input it produces the tree and the diagnostics this spec defines.

- **A conforming processor** implements stages 1 to 4b and produces the expanded tree.

Optional features are claimed separately, each with its own test cases:

Feature	What it adds
a host language	running components defined in that language (`js`, `ts`, `jsx`, `tsx`, `svelte`), one per claim; a component that throws while rendering is W118
component analysis	checking each host-language component against its signature fence (W104, section 7.1)
remote data	loading `https://` data (section 8.6)
interactive inputs	readers change inputs, and the processor renders again (section 8.7)
origins	`origin` on every expanded node, for editors (section 12.3)

A claim names the class and its features, for example "compmd/1 processor with jsx and remote data".

What a processor that renders **MUST** do (render fallbacks, escape interpolated text, show inputs statically when it isn't interactive) is required by this spec, but not tested by the conformance set.

Layout. The meaning of each layout param is the CSS property it names (section 9). Conformance is tested on the expanded tree, not on output. A reference HTML and CSS rendering of the built-ins, `@compmd/render`'s (README), is informative.

13.2 The conformance set

Each case is a folder, since imports and data need several files:

```
conformance/<area>/<case>/
  case.json          what the case needs: {"requires": ["processor",
"remote-data"]}
  input.comp.md       plus any imported documents and data files
  context.json       optional: render time, time zone, input values, host
metadata, URL-to-file map, pending sources
  parse.json         the expected tree after stage 1
  expanded.json      optional: the expected tree after stage 4b
  diagnostics.json   optional: the expected diagnostics, [{code, file,
line, column, related}]; none when absent
```

- Trees are compared without their `position` and `closePosition` fields, unless `case.json` sets `"positions": true`, and without `origin`, unless the case requires `origins`. Diagnostics are always compared by code and position; messages are not compared, and order doesn't matter. A diagnostic in another document of the case (an import, a pack) names it in `file`, relative to the case folder, or as `@scope/pack@version/path` for a pack (section 10.4).
- `context.json` maps remote URLs to local files, so remote-data cases run without a network.
- A case for interactive inputs **MAY** list several contexts in order, each with its expected expanded tree; each must equal a fresh render with that context (section 8.5).
- Every example in this spec is also a case.
- Every **MUST** in this spec belongs to exactly one class or feature, and every diagnostic code and every section has at least one case. A case for a host language may expect W118, found when rendering: a processor that claims the language runs the component to report it. A test runner reports what is not covered.

---

14. Degradation contract

A CompMD document rendered by a plain CommonMark or GFM renderer **MUST** keep all author content readable. Conformance test suites check these rules against the CommonMark and GFM reference renderers.

1. CompMD syntax never hides author content: everything inside a container renders as normal Markdown.
2. Every CompMD construct is one line, so a plain renderer shows at most one literal line per construct, never a block of code or a broken table.
3. Component lines never change how the surrounding Markdown parses: no significant indentation, no syntax that opens a code block or HTML block.
4. Leaf and inline components carry readable fallback text.
5. Definitions show as syntax-highlighted code, or live in imported files.

Generated content is exempt. Output produced by `each`, `if` or `value` from data cannot be readable in a plain renderer, which shows the template once (for example `- {{c.text}}`). Authors who need plain-renderer readability write that content statically.

What a plain renderer shows:

CompMD construct	Plain rendering
`::: card {title="Setup"}`	one literal text line
container body	rendered Markdown, unchanged
`:: #footer`	one literal text line
`::: /card`	one literal text line
`:: chart[Weekly installs] {…}`	one literal line containing the fallback
`:badge[beta] {tone=info}`	literal text, fallback visible
`{{md.doc.version}}`	literal text
`component=` definition fence	highlighted code block
frontmatter	a metadata table (GitHub), hidden (renderers that read frontmatter), or a horizontal rule and a heading (plain CommonMark)

Authors **SHOULD** separate component lines from surrounding paragraphs with blank lines. Otherwise CommonMark joins a component line to the paragraph above it, which can change the structure around it (two lists become one), and processors report W108 (section 3.2).

15. Security

Code from a host-language component runs at one of three **trust levels**. The processor's configuration sets the level of each pack; a document can never raise one.

Level	Where the code runs	What it reaches	For
Strict sandbox	in a sandbox	only its contract (the rules below)	the default: code from documents, and from every pack the configuration gives no other level

Level	Where the code runs	What it reaches	For
Permissive sandbox	in a sandbox	its contract, plus the network to the origins the configuration lists for its pack, and `host` (section 7.4)	packs the configuration makes permissive, such as a map that loads its tiles
Trusted	where the processor's own code runs, such as the host application's page	what the processor's own code reaches, and `host` (section 7.4)	packs built with the host application, which the processor's configuration names by their place in the application, never a registry package

- **Every component's code but trusted code MUST run sandboxed**, strict or permissive, wherever it runs: in a reader's browser, on a build server, in an application. This applies to inline definitions and to imported ones, so code from a document or a registry pack always does.
- **Trusted code is the host's own:** built, reviewed and shipped with the application, as the processor's own code is, so a sandbox would protect the application from itself. Its params and slots still come from documents, which may be someone else's, so it **MUST** treat them as untrusted input: no HTML from a param, URLs checked as interpolation checks them (section 8.1), and actions outside the document only as for components given `host` (below).
- `compmd` definitions run no code: they only expand Markdown, params and slots, so they need no sandbox.
- Interpolated text values **MUST** be escaped for the output format; they are never parsed as Markdown or HTML.
- A processor **MUST NOT** fetch remote data unless the user or host configuration allows that origin; a document can never grant itself access (section 8.6). Packs come only from the registry the processor is configured with, and only those its configuration allows (section 6.4).

The sandbox. The spec states what a sandbox guarantees, not how it is built. Code in a strict sandbox:

1. **Sees only what the host contract gives it** (section 7.4): its params, its slots, its target element and `set` for its bound inputs. Not the rest of the document, its data, or the page around it.
2. **Has no network**: no requests, images, fonts, style sheets or sockets from anywhere. Its code and assets come from its pack, its binding's runtime and the libraries the processor

provides it (section 7.6). Data reaches a component only as params, from data imports (section 8.6), which are declared in the frontmatter and cannot carry `md.data` or inputs the component can change (F031).

3. **Can't reach the reader's or the host's session**: no cookies, storage, credentials, host APIs, file system, environment variables or processes.
4. **Can't navigate or open windows.** A link in a component doesn't open on its own; a processor **MAY** offer to open it after showing the reader the full URL. Links written in Markdown are unaffected.
5. **Can be stopped**: a processor **SHOULD** stop a component that runs too long and render its fallback (W118).

A permissive sandbox keeps rules 3 to 5 and adds to rules 1 and 2 only what the configuration lists for its pack: `host`, and the network to those origins (a map component loading its tiles), never because a document or a reader prompt asks for it.

- Components of one document **MAY** share a sandbox, and so **MAY** the documents a processor shows together as one application, when it gives all their code the same trust. Code it trusts differently runs in a separate sandbox, and so does each permissive pack, so the network one pack is given reaches no other code. The boundary is between code of one trust and everything else, not between components.
- **Packs the configuration doesn't list** run in a strict sandbox per document: documents shown together don't share one for them, since a processor can't vouch for packs it never saw. So a pack one document imports can't read what another document's components are given.
- What a sandbox can't stop is what a component draws in its own box, such as a fake password prompt. What is typed into it has no way out (rules 2 and 3), and a processor **SHOULD** show a sandbox's box as the document's content, never over its own interface.

Components that act with someone's access. A component given `host` (section 7.4), and a trusted one, acts with someone's access, but the document that invokes it, and so its params, may be someone else's: a shared document can aim such a component at what it shouldn't touch.

- **Acting:** such a component **MUST** act outside the document (send, write, share, pay) only on the reader's action, after showing the reader what it will do: "Send *accounts.csv* to attacker@evil.example?". A processor with no reader present, such as a build, gives no such action. In documents that come with the component's pack (an application's own screens), the component acts within what the configuration grants: whoever gave the pack its level trusted those documents with it. In any other document, the rule above applies.
- **Reading for others:** a processor that renders for people other than those whose access its components use (a site built with its operator's credentials and published) **MUST** give `host`, and render trusted components, only in documents it trusts, such as the site's own

pages and not a contributor's change. Rendering for the reader with the reader's own access needs no such rule: the reader sees only what they may see.

- A component in a strict sandbox gets only the contract and no network (rule 2), so private data passed to it as params has no way out.
- Without rule 2, a component that is passed private data and allowed to reach its own origin "for the weather" can send the data with the weather request; a reader can't tell the two apart. With it, private data has no way out of the sandbox.
- *Informative:* in a browser, a cross-origin iframe with `sandbox="allow-scripts"` and a content security policy that allows only the processor's runtimes and libraries and the pack's own files; on a server, an isolated JavaScript runtime with no I/O.

Privacy. A fetch made while a reader views a document tells the origin the reader's address, the time, and any input values in the URL, as an image in any Markdown does. A document rendered once at build time reveals no reader. For fetches at view time:

- The reader decides which origins a document may reach, the first time each is needed (section 8.6); the document's author cannot.
- Processors **MUST NOT** send cookies or other credentials with document fetches, except those the host configured for that origin, and **SHOULD** send no `Referer`.
- Processors **SHOULD** share cached copies between readers when the URL is the same, so a cache hit tells the origin nothing.
- Packs are fetched by the processor, not the reader's browser (section 6.4), unless the processor's configuration lets the browser fetch them from the registries it names. A registry the browser fetches from learns the reader's address, the time, and the names and versions of the packs the document imports; never the document, what it shows or its inputs. Such fetches send no credentials, and the processor **SHOULD** tell its readers.

---

16. Example

Release notes as an author would write them. The sections above show each construct on its own. A complete example that uses every construct, with its data files, is in `examples/complete/`.

```
---
md:
  version: 1
  import:
    - from: "@myorg/ui@1"
      names: [card, badge]
```

```
    - from: ./chart.jsx
      names: [chart]
title: Release notes
version: "1.4"
changes:
  - {text: "New wgpu renderer", pr: 412, breaking: false}
  - {text: "Config format 2", pr: 398, breaking: true}
---

# {{md.doc.title}} :badge[beta]{tone=info}

RayPlay {{md.doc.version}} ships a new renderer.

```compmd component=callout {padding=m border-radius=m border=subtle
background=subtle class=("callout-" + kind)}
params:
  kind: tip | warning = tip
slots: [default]
---
{{md.slot}}
```

::: grid {gap=l}
::: + md:grid-template-columns="2fr 1fr"

::: card {title="Setup" icon=rocket}
Install with `cargo install rayplay`, then run `rayplay init`.

:: #footer
Takes about 2 minutes.
::: /card

:: chart[Weekly installs: up 18% since 1.3] {src=installs.csv type=bar}

::: /grid

What changed

::: each {in=(md.doc.changes) as=c join=list}
- {{c.text}} ([#{{c.pr}}](https://github.com/acme/rayplay/pull/{{c.pr}}))
::: /each

::: if {when=(md.doc.changes.exists(c, c.breaking))}
::: callout {kind=warning}
```

```
The 1.3 config format is no longer read. Run `rayplay migrate` first.
::: /callout
::: /if
```

In a plain renderer, this shows the title and badge text, the release sentence, the definition as a code block, the card, chart and callout text in order with one literal component line around each, and the change list's template once (section 14).

---

17. Open issues

- ☑ **Frontmatter (decided, section 2):** one `md` key holds everything CompMD reads (`version`, `import`, `export`, `data`, `inputs`, `lang`, `tz`), and every other key is the author's (`md.doc`); a `.comp.md` file without the key is version 1; host metadata is equivalent to frontmatter, without merging; YAML 1.2 core without tags, anchors or duplicate keys; minor versions only add. Replaces the earlier `flavor: compmd/1` marker, which mixed CompMD's keys with the author's.
- ☑ **Names (decided, sections 3.1, 6.3):** local names shadow `@compmd/core` (W112) except the keywords `if`, `each`, `value` and `box`; a wildcard and core providing the same name is ambiguous when used (F023); kebab-case by convention, `_` allowed; slot content resolves in the caller's document; processors **MAY** provide ambient packs.
- ☑ **Markdown output (decided):** none in version 1. A CompMD document stays readable in other Markdown renderers (section 14); processors don't define a plain-Markdown rendering of it.
- ☑ **Live documents (decided, section 8.5):** rendering is a function of the document, its imports, its data and the render context; processors choose when to render again.
- ☑ **Inputs (decided, section 8.7):** typed document inputs with defaults, input components bound to them, one direction from inputs to output.
- ☑ **Diagnostics and recovery (decided, sections 4.5, 10):** a recovery rule for every code, from four principles; container fallbacks keep named slots; errors in imported definitions point at the definition, with the invocation chain; a JSON form that maps onto LSP.
- ☑ **Alternate spellings (deferred past version 1):** an HTML-comment spelling of the same tree would be a second grammar for every construct, against the precision goal; a mapping to Pandoc divs is a conversion a tool can make from the document tree, not part of the format.
- ☑ **Grammar precision (decided, sections 3, 7.3, 8.1):** value kinds by first character; CompMD containers nest with CommonMark containers; `{{ }}` lexed like code spans,

never producing a URL scheme; `-` in param names written `_` in expressions; signatures and inputs as YAML with short and long declarations; data keys renamed in `schema`.

- ☑ **Base syntax (decided, section 2.3):** CommonMark, GFM as GitHub renders it (with footnotes), alerts and math in compmd/1; wiki links opt-in through `md.syntax`; diagrams are code blocks; processors add no syntax; raw HTML parsed as CommonMark says and rendered safely.
- ☑ **Processing, tree and conformance (decided, sections 11–13):** five stages with parsing independent of imports and data; an mdast-based tree; an expanded tree that is a renderer's whole input (input-bound params carry their value and input, the root lists the inputs); every block component is one box, a `compmd` component's styled by params on its definition fence and overridden by the caller's as a later CSS cascade layer, with `display=contents` for a component without one; parser and processor classes with optional features; conformance tested on trees, layout meaning defined by CSS.
- ☑ **The `md.` namespace (decided, section 8.1):** every variable and function CompMD provides is under `md.`; there is no `$` sigil.
- ☑ **Expressions (decided, sections 8.1, 8.2, 8.5):** numbers are CEL `double`s, with mixed `int` and `double` arithmetic and CEL's integer division kept; a `date` is an ISO date string, times are CEL timestamps, with `md.addDays`, `md.daysBetween`, `md.startOf` and `md.dateOf`; an exact function list (CEL standard, the strings extension, CompMD's formatting, date and list functions; no `matches`); locale data from an unpinned recent CLDR, `md.plural` with CLDR categories; how `{{ }}` writes values that aren't strings, a map with a `name` as its name; a 100,000-node expansion every processor renders (F034 beyond).
- ☑ **Host-language components (decided, sections 7, 7.4, 7.6):** one host contract (params, slots as lists of items rendered up front, `update` and `destroy`, an outer box the processor owns), bindings for React and Svelte 5 in version 1, `runtime=<framework>@<major>` after the source language; a component changes an input only through a param of type `input<…>`; a signature fence in a processor's pack marks a component the processor implements (no `native` language).
- ☑ **Code renderers (decided, sections 2.3, 6.3, 7.3, 7.7, 12.3):** a leaf whose signature `renders` a language, or names of one, renders the code blocks of that language, the block's text its `code` param; one component for both forms, so the host contract is unchanged; params on the info string, read only for rendered blocks; `alt` as the text alternative; resolution by language, F023 for two renderers at one level; the code block is the fallback; no inline form; `component=` only on definition languages.
- ☑ **Template slots (decided, section 7.5):** a slot rendered once per element of an array param, binding a name the author picks (a `name` param); host components get `{value, items}` instances; one level; no `expression` type. `each` is declared the same way.
- ☑ **General events (deferred past version 1):** components emitting events that documents route to inputs or hosts. Inputs bound through `input<…>` params cover version

1's interactive documents; events can be added in a minor version without changing them.

- [x] **Vue binding (deferred past version 1):** a later minor version adds it; nothing in the host contract blocks it.
- [x] **Control flow (decided, section 8.3):** `if` and `each` are ordinary `@compmd/core` containers, with `else`, `before` and `empty` slots and join modes.
- [x] **Signatures for host components (decided, section 7.1):** exported host-language components need a signature fence (F029); local ones **MAY** derive theirs from the component (W105 for what can't be typed).
- [x] **What hosts give their own components (decided, sections 7.4, 15):** a processor **MAY** give components from permissive packs more than the contract, in `host`; such components, and trusted ones, act outside the document only on the reader's action, and a processor rendering for others gives `host`, and renders trusted components, only in documents it trusts.
- [x] **Sandbox (decided, section 15):** requirements, not a mechanism: a component sees only its contract, has no network (data comes only through data imports), no session, no navigation, and can be stopped; one sandbox per document.
- [x] **Trust levels (decided, sections 6.3, 7.1, 15):** three, set per pack by the processor's configuration: a strict sandbox (the default), a permissive sandbox (the network to listed origins, and `host`), and trusted (the host's own code, built with the application, outside a sandbox, its inputs untrusted). A signature fence alone in an ambient pack declares a component the host implements; an ambient pack's name never resolves to a registry package.
- [x] **Layout (decided, section 9):** params named after the CSS property they set; `box`, `stack`, `row` (wrapping), `grid` (cards by `min`, or exact with `grid-area`) and positioned layout: any block placed in its box with `inset-*` (no canvas component, no `z-index`); spacing as tokens or lengths; size, padding, order, self-alignment, `aspect-ratio`, `scroll`, `fill`, `sticky`, `show` and `hide` on any component; logical directions only; breakpoint prefixes `sm:`, `md:`, `lg:` at fixed 480, 768 and 1024 px, from that width up, following the block's container; a closed vocabulary, each param meaning its CSS property, which other outputs follow as far as they can.
- [x] **Visual styling (decided, sections 9.2, 9.4):** `background`, `border` and `border-radius` on any component take a token (`subtle`, `strong`, `accent`; `s` to `full`) or a CSS value, plus `class` and `id`. The tokens are custom properties with defaults built from the text color, so a document looks right in light and dark mode without a stylesheet.
- [x] **Accessibility and direction (decided, sections 2.1, 5.1, 7.2, 7.4, 9.2):** output order is source order whatever the layout; hidden blocks are hidden from assistive technology; host components get their fallback as text alternative; headings in `compmd` definitions are relative unless the signature says `headings: absolute`; `md.dir` derived from `lang`; `lang` and `dir` on any component.

- [x] **Markdown in data (deferred past version 1):** data values are text, never Markdown (section 8.1), so data from a remote source can't add links, images or HTML to a document. A `markdown` schema type, with rules for inline content and link schemes, can come in a minor version.
- [x] **Smaller points (decided, sections 2, 3.3, 5.1, 5.2, 8.4):** continuation lines are the only multi-line params; no param defaults inherited by a subtree; named closers only, no bare `::: /`; no Quarto-style layout arrays; a formatter changes whitespace and blank lines only; `class` and `.name` add up; the `variant=compmd` media type is registered when version 1 is published; `value`'s `of` is a string.
- [x] **Packages (decided, sections 6.1, 6.4, 15):** `@scope/pack` is an npm package, its document named by `package.json`'s `compmd` field; version ranges, with an optional `integrity` hash on exact versions (F035); packs pin their own imports; no lock file in the spec; `@compmd/*` is reserved and never fetched. A document may import any pack by default, in the strict sandbox, one per document for packs the configuration doesn't list; a processor may allow only listed packs, deny packs as its own policy, and cap their size (F026); it gets packs itself, never the reader's browser; a pack's built module is loaded from its checked files.
- [x] **Origins (decided, sections 12.3, 13.1):** an optional `origin` on every expanded node, the range of this document that made it with its occurrence among siblings of the same range; a definition's body maps to its invocation. Keys for matching nodes across edits are the editor's, not part of the format, since they depend on the text before the edit as well as after it, and stage 4b is a function of the document (section 8.5).
- [ ] **Record keys (open):** `each` iterations and template slot instances are told apart by their index, not by their record, so an editor keeps a component's state with its place in the list: with a counter per item in `[Apples, Pears]`, adding `Kiwis` at the front gives Kiwis the clicks Apples had. A `key` naming each record (an expression on `each`, a param named by a template slot's signature) would let state follow its record. To decide when a host needs it.
- [x] **Data imports (decided, sections 8.6, 15):** data from files (YAML, JSON, CSV per RFC 4180 with a `delimiter`), packs, `https://` URLs and `host:` sources, the one extension point for hosts, which add no variables; relative paths resolve against the file they're written in, never above its root; a reader decides the origins a document reaches when it is viewed, asked the first time; fetches carry no credentials. Decimal commas are not read as numbers.

CompMD packs on a registry, version 1 (draft)

A companion to the CompMD specification, for pack authors and for hosts that fetch packs from a registry. The spec says what a pack is to a document (§6.4); this document says what a pack published to an npm registry must be, how a host reads one, and what a library pack is. It is versioned with the spec: version 1 of each goes together, and every change to it has an entry in the changelog.

Last changed 2026-11-02.

1. Conventions

- The key words **MUST**, **MUST NOT**, **SHOULD** and **MAY** are used as in RFC 2119.
- A **registry** is a server that speaks npm's registry protocol: npm's own, a mirror, a private registry or a proxy. A **host** is the application whose processor renders documents; its configuration names its registries (section 4).
- A **registry pack** is a pack published to a registry as an npm package (spec §6.4). A **library pack** is a package that holds a library packs share (section 5); it has no document.

2. A pack on a registry

A registry pack is a CompMD pack as spec §6.4 says: an npm package whose `package.json` is its manifest (F022 when `md.export` disagrees with it), whose document the `compmd` field names, and whose integrity is the hash the registry publishes for its tarball. In addition:

- **Its name is scoped**, `@scope/name`, as every pack's is; `@compmd/*` is reserved. No prefix is required: an import always names a pack in full, and the field and keyword below say what it is. A scope that publishes a library and a pack of the same name **MAY** prefix the pack's with `compmd-` (`@acme/charts`, a React library, beside `@acme/compmd-charts`, its pack).
- **It says it is a pack**, in its `package.json`:
  - the `compmd` field **SHOULD** be written, even when it names the default document (`"compmd": "index.comp.md"`). A registry keeps every field in a version's manifest, so a tool can tell a pack from other packages without its tarball. A host still reads a package without the field as spec §6.4 says.

    - its `keywords` **SHOULD** include `compmd-pack`, which a registry's search indexes, so a search for `keywords:compmd-pack` lists packs. Only packs use this keyword; packages that help build packs use `compmd`.
- **Each built module SHOULD be one file**, with no relative imports, so it loads where it is passed as text (spec §6.4). Its imports are only its binding's runtime and the libraries its pack names in `peerDependencies` (spec §7.6); every other library it uses is bundled in. A pack whose modules import files of their own is still a pack, but a host that can't load them renders those components' fallbacks with W121.
- **Its `dependencies` are never installed** by a host. What its components use is bundled into their modules, and the packs it imports are its document's `md.import`, which the host resolves (section 3). `dependencies` **MAY** still list what its build used.
- **Its `files` SHOULD** leave out what a host never reads (sources, tests), so its tarball is small. A host reads only `package.json`, the pack's documents and the files they import.
- **Its size** counts against the host's cap, if the host sets one: the version's unpacked size, which the registry's metadata states (section 3).

3. How a host reads a registry pack

A host that fetches a document's packs from a registry resolves them before processing, as it fetches the document's data:

1. **Which packs.** The packs the document imports that the host doesn't hold, and then those the fetched packs' documents import, until none is missing. A pack's imports go through the same host's registries and policy as the document's: a pack can't name a registry.
2. **Which version.** A range resolves to the newest version in the registry's list that matches it (spec §6.1). None matching is F026.
3. **Policy first.** Before anything is downloaded, the host applies its configuration (spec §6.4): a pack it doesn't allow or denies is F026; so is a version whose unpacked size, from the registry's metadata, exceeds its cap.
4. **The tarball, checked.** The host downloads the version's tarball and **MUST** check it against the integrity the registry's metadata gives for it (`dist.integrity`) and, when the import pins one, against the import's `integrity` (F035 when either doesn't match). Nothing in the pack is used that the tarball's hash doesn't cover.
5. **Its files.** The tarball is a gzipped tar of the package under `package/`. The host reads `package.json` from it (never from the registry's metadata alone), the document its `compmd` field names, and the files the pack's documents import. A built module is loaded from these files, by its place in the pack (spec §6.4).

A version's tarball never changes, so a host **MAY** keep what it fetched by its integrity, and serve a pinned import from that cache without fetching. A host **MAY** keep a pack's version

list for as long as the registry says it may be cached.

4. Who fetches

Which registries a host uses is its configuration, by scope with a default for every other scope, as npm's `.npmrc` does. A document never names a registry, and neither does a pack, so neither can make a host or a reader's browser contact a server of its choosing. Without a registry in its configuration, an import of a pack the host doesn't have is F026.

- **By default, the host fetches**, itself or through its own proxy or cache, so the registry learns nothing about who reads which document (spec §6.4, §15 *Privacy*).
- **The reader's browser MAY fetch directly**, from the registries the host's configuration names, when that configuration says so; a host with no server, such as a static site, may need to. The registry then learns the reader's address, the time, and the names and versions of the packs the document imports, never the document or what it shows. The host **SHOULD** tell its readers. Such requests **MUST** send no cookies or other credentials, and carry nothing from the document but the names and versions of its imports. A document can never choose this; only the host's configuration can.

5. Library packs

A library pack holds one library that packs share, so a sandbox loads one copy instead of each pack bundling its own: worth it for large libraries several packs use (three.js, KaTeX).

- **It says it is one** with an object in the `compmd` field: `"compmd": { "library": "dist/index.js" }`, the path of its one module file. It has no document and no components. Its name is scoped, like a pack's, and **SHOULD** say what it wraps, with the library's version as its own (`@kit/three@0.170.0` is three.js 0.170.0).
- **Its module is one file**, an ES module whose only imports are the runtimes and other libraries by name, never by relative path.
- **A pack uses it** by naming it in `peerDependencies` (`"@kit/three": "^0.170"`) and importing it by that name in its components' code. A host that offers no library of that name (spec §6.3) resolves it from its registries like a pack (section 3), under the same policy. A library pack uses other libraries the same way, by naming them in its own `peerDependencies`.
- **One version per document.** For each library, the host uses the newest version that matches every range the document's packs ask for. When no version matches them all, each range gets its own, unless the library says it can't have two copies in one sandbox: `"compmd": { "library": …, "single": true }`. Then the version is the one the first pack to ask gets, in the order the document's imports resolve, and each pack whose range it doesn't match gets W121 on its import, naming the library, both ranges and the version the document has (spec §6.4); its components that don't use the library still render.

- **A library that can't be had** is W121 on the import of the pack that names it, saying why: the host denies or refuses it, or doesn't allow it under its configuration's `imports`; no version matches; the package isn't a library pack; or libraries name each other in a cycle. The pack's components that don't use it still render.
- **It runs at the level of the pack that imports it**, in that pack's sandbox, and adds no trust of its own (spec §15).