
Guidedog Discussions

Design, rationale, and review

01 / SOURCE

Write with meaning.

02 / STRUCTURE

Make relations clear.

03 / PUBLICATION

Read on screen and paper.

Contents

Work on a discussion	3
Create and inspect	3
Review and promote	3
Record a decision	4
Publish and recover	4
Discussion registry	7
Discussion reading order	9
The Guidedog Discussion Process	9
Guidedog: CLI and documentation toolchain	19
Guidedoc: Odin conversion engine	32
Memory budgets and what a build may read	57
An Odin domain and Odin API documentation	66
Sphinx-style projects and recoverable publication	71
Historical provenance	87
The Zen Discussion Process	87
Historical registry	94
Historical export manifest	100

A design record explains a decision and its consequences. It names the assumptions, the alternatives, and the evidence.

Read the state first. A proposal is not an accepted design. A Git commit is not the GDS state committed.

This is a separate documentation project from the manual. The manual teaches use. These records explain the design.

Work on a discussion

A GDS explains one decision. Start with the problem and the proposed result. State the assumptions. Give a worked example. Name the alternatives. Keep the implementation status separate from the proposal.

Create and inspect

```
guidedog gds new "Cache policy" --author="Ada" --labels=build
guidedog gds show cache-policy
guidedog gds list --state=prediscussion
```

The draft has an XXXXX number. Its filename ends in `.rst`. Metadata occupies the first field list. Authors and labels are JSON arrays. Dates are real ISO calendar dates. Keep the title and the explanation in the record.

```
:gds-number: XXXXX
:gds-title: Cache policy
:gds-state: prediscussion
:gds-type: Standards Track
:gds-authors: ["Ada"]
:gds-created: 2026-10-01
:gds-updated: 2026-10-01
:gds-discussion: Local review
:gds-labels: ["build"]
```

```
Cache policy
=====
```

```
Abstract
-----
```

```
State the decision and its observable result.
```

A record needs an Abstract and References section. Use RST tables, Graphviz diagrams, and math directives where they clarify the reasoning. Do not hide normative prose in a typesetting include.

Review and promote

```
guidedog gds check
guidedog gds promote cache-policy --dry-run
guidedog gds promote cache-policy
```

Promotion assigns a permanent number and opens discussion. It does not accept the design. Numbered records are not renumbered or reused. The CLI updates the registry and reading order in the same recoverable transaction. Review prose links after a filename changes; promotion does not rewrite them.

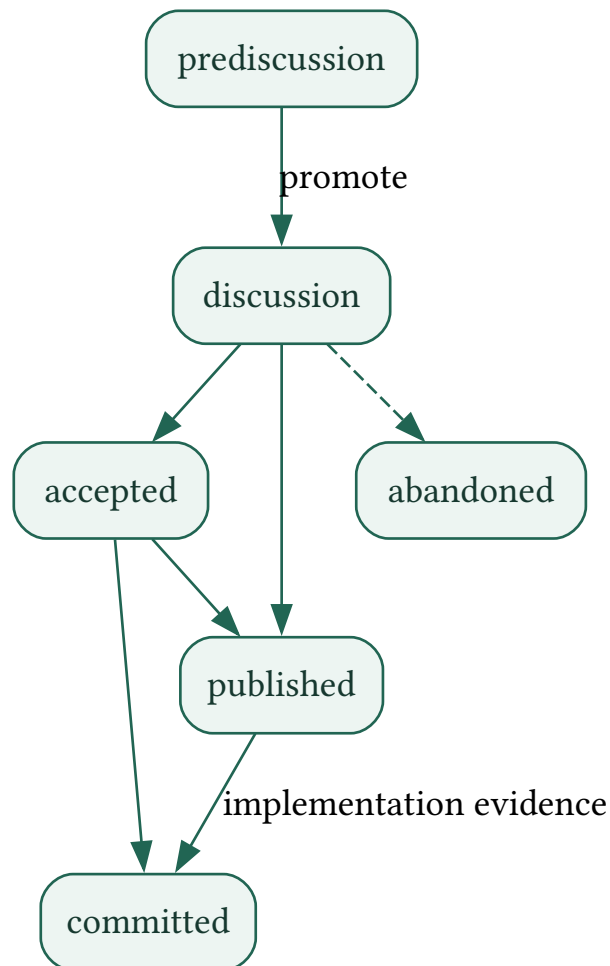


Fig. 1 A number identifies a record; a state expresses its review result.

Record a decision

```
guidedog gds state 4 --to=accepted --reason="Review converged"
guidedog gds state 4 --to=committed --reason="Implemented" --evidence="Tests
and review"
```

A committed state requires evidence. The source history is not evidence by itself. Do not claim another person's approval. The maintainer owns the state decision.

Publish and recover

```
guidedog gds index
```

```
guidedog gds build --target=both
guidedog gds build guidedoc-odin --target=pdf
guidedog gds recover --rollback
```

HTML builds the whole discussion site. A selector limits separate PDFs. A selected PDF preview uses `_build/selected/<artifact>/pdf`. It leaves the complete catalog in `_build/pdf` intact. The project outputs live in `_build/html` and `_build/pdf`. An interrupted metadata edit needs explicit rollback. Recovery refuses a later edit that conflicts with its journal. Inspect the conflict before retrying.

Read [The Guidedog Discussion Process](#) for the process contract. Read [Guidedog: CLI and documentation toolchain](#) for the CLI contract.

Reserved through: 0

Discussion registry

Table 1 Records and review states

GDS	Discussion	State
0001	The Guidedog Discussion Process	committed
0002	Guidedog: CLI and documentation toolchain	published
0003	Guidedoc: Odin conversion engine	committed
0004	Memory budgets and what a build may read	committed
0005	An Odin domain and Odin API documen- tation	committed
0006	Sphinx-style projects and recoverable publi- cation	committed

Discussion reading order

Number: 0001
Title: The Guidedog Discussion Process
State: committed
Type: Process
Authors: Vikrant Rathore
Created: 2026-09-29
Updated: 2026-10-11
Discussion: Local review; no external discussion URL assigned
Labels: process

The Guidedog Discussion Process

Abstract

Guidedog Discussions (GDS) are the project's RFC/RFD-style specification and decision records. The subject is Guidedog, a Sphinx-like documentation system built in Odin. This revision adapts the lifecycle from ZDS 0001 and preserves its lifecycle and numbering process, and adds explicit PEP-inspired guidance for rationale, compatibility, review history, and implementation evidence. GDS is a collection of standalone discussions, not a book divided into chapters.

Introduction and scope

Guidedog builds a documentation project into a responsive HTML site or a PDF book using Typst. Its scope includes navigation, cross-document references, directives and roles, themes, project configuration, incremental builds, local preview, and single-binary distribution. Guidedoc is one subsystem: its conversion library. GDS decisions govern the whole product, not a renamed zenfmt file converter.

A discussion explains one coherent proposal and the reasoning that makes it worth adopting. It can specify a feature, a format mapping, a process, or an assessment. A future user manual belongs outside GDS. The HTML index connects standalone records; each record has its own PDF. There is no PDF index or combined PDF build.

The process retains ZDS 0001's local placeholder drafts, maintainer-assigned numbers, state transitions, registry, standalone reStructuredText records, and

format-record obligations. RFCs inform memo structure and considerations; Oxide RFDs inform repository-based discussion; PEPs inform focused proposals, rationale, alternatives, and the distinction between editorial readiness and technical acceptance. We adopt these practices without claiming the governance or authority of those projects.

This process is implemented by `guidedog gds` for `reStructuredText` records. The register and lifecycle events record the maintainer-requested reconciliation of implementation status. The inherited process alone is not acceptance evidence. Its original wording remains in `legacy/0001-zds-process.rst`.

When a Guidedog discussion is needed

Write a GDS for user-visible command or configuration changes; document graph and reference semantics; RST directives, roles, CommonMark extensions, or Typst behavior; HTML and PDF builders; theme contracts; plugins and trust boundaries; cache invalidation; memory and CPU limits; packaging; and project governance. A proposal explains effects on existing documentation projects and on migration from Sphinx where relevant. State whether a Sphinx feature is supported, adapted, or out of scope. Sphinx compatibility is not inferred from RST support alone.

Bug fixes preserving specified behavior, routine refactors, and additional tests usually amend an existing record or need no new record. The future user manual explains how to use Guidedog; release notes state what shipped; GDS explains what was decided and why. Do not move normative design decisions into manual chapters.

Terminology

A **placeholder draft** has number `XXXXX` and state `prediscussion`. An *assigned GDS** has a permanent four-digit number. The **author** maintains the proposal and records feedback. A **maintainer** assigns numbers and records project decisions. The **registry** describes available discussions and historical references. A **format record** specifies parsing, mappings, policy refusals, and fidelity.

Discussion lifecycle

New proposals begin as `records/XXXXX-<slug>.rst`. Copy the template, fill in metadata, and iterate locally. Placeholder drafts appear after numbered records in the index. When ready for review, a maintainer assigns the next unreserved number, renames the file, and updates metadata, registry, and bundle together. A number is an identifier, not evidence of approval. It is never reused.

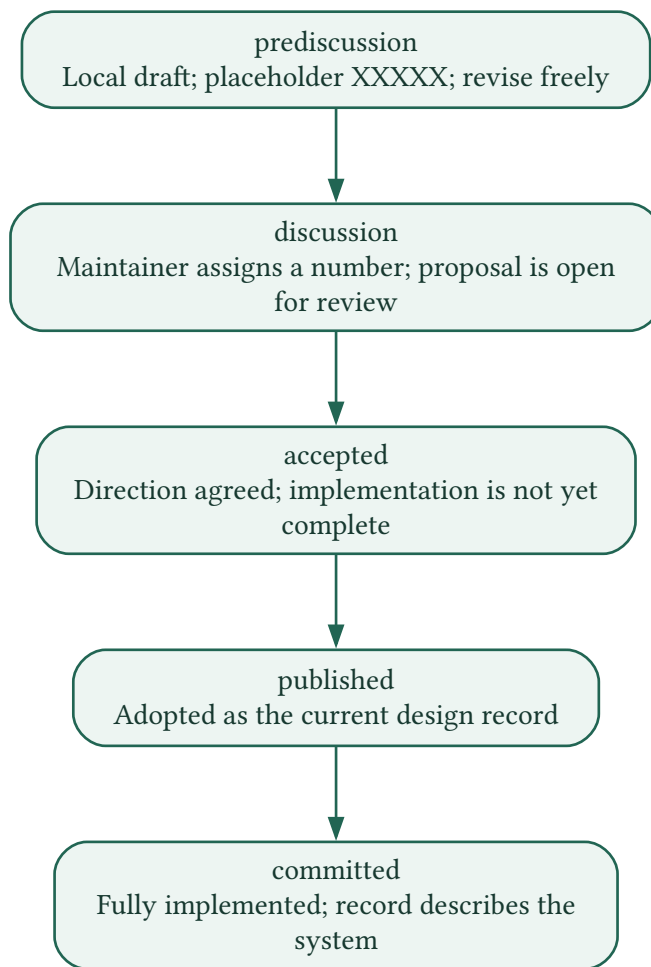


Fig. 2 The usual forward path. The transition table defines shortcuts and abandonment.

Transition	Meaning and responsible actor
prediscussion → discussion	Maintainer assigns the number and registers the record.
discussion → accepted	Maintainers agree on the proposed direction.
discussion → published	Maintainers adopt a document whose text is the deliverable.
accepted → published	Maintainers adopt the design record while implementation continues.
accepted / published → committed	Maintainers verify implementation evidence.
Any pre-committed state → abandoned	Maintainers record why the proposal was closed.

These are the six inherited states. Do not replace them with PEP states such as Final, Active, or Rejected. A Git commit does not mean the GDS state is `committed`. A successful Typst compile does not mean the design is accepted. Preserve

numbered abandoned records. For a committed design superseded later, retain its history and link a successor instead of pretending it was never implemented.

Metadata and record structure

Each file starts with one `:gds-name:` metadata field per line. Required names are `number`, `title`, `state`, `type`, `authors`, `created`, `updated`, `discussion`, and `labels`. Scalar values are plain text. Authors and labels are JSON arrays of strings. The block is data; it is never evaluated as code. Published headers use `Number`, `Title`, `State`, `Type`, `Authors`, `Created`, `Updated`, `Discussion`, and `Labels`. Authors and labels appear as readable lists. This presentation does not rename the source fields or change what the GDS CLI reads and edits. The type is `Standards Track`, `Process`, or `Informational`; it does not change the inherited state machine. Dates use ISO format. An unavailable discussion link is stated plainly; never fabricate an issue, author contact, or acceptance decision.

An assigned number must match its filename. A placeholder is identified by its slug, so multiple `xxxxx` drafts do not collide. Related, superseding, and superseded records are named explicitly in prose and registry metadata when relevant.

A record normally includes abstract, motivation, scope, specification, rationale, security and operational considerations, backwards compatibility, alternatives, open questions, reference implementation or implementation status, acceptance criteria, review history, and references. Headings may fit the subject. A section that does not apply says why; avoid boilerplate that adds no information.

The specification states observable behavior and testable invariants. Rationale explains choices; it is not a second ambiguous specification. Examples illustrate the contract and do not silently weaken it. If uppercase `MUST`, `SHOULD`, or `MAY` is used normatively, cite RFC 2119 and RFC 8174 and explain their use. Ordinary precise English is preferred where no formal requirement keyword is needed.

One discussion lives in one `reStructuredText` source file. Shared templates are allowed. Normative prose stays in the record. Graphviz diagrams and math directives are ordinary source content, so the project engine can check and render them. Split an oversized proposal into focused discussions with explicit dependencies. The template is a starting point rather than a demand for a particular narrative order.

Format records

Retain the three ZDS 0001 requirements: a mapping table for each recognized source construct and its semantic representation; deliberate omissions with reasons; and round-trip expectations. Because this project currently has only HTML and PDF outputs, round-trip recovery may be unavailable. Say so rather than implying that PDF can reconstruct source. Distinguish parsing coverage, target representation, policy refusal, intentional omission, and unfinished work.

Full RST requires a coverage ledger for the specification, standard directives, and roles. CommonMark names the pinned specification and suite. Typst behavior names the pinned toolchain. Every mapping change updates its fixtures and record. Changes to shared AST semantics require a dedicated GDS; localized mapping corrections may amend the existing format record.

Coding and writing rules

Every new source file has at most **1,408 physical lines**. A line should remain within **99 characters** and must not exceed **108 characters**. Source is Odin and every other implementation language shipped with it: CSS, JavaScript, HTML templates, shell, Rust, C, and TOML. Documentation (Markdown, reStructured-Text, and Typst prose, diagrams, and markup structure, these records included) is written for readers and exempt from every one of these limits: line width, file length, and procedure length apply to code only. Every procedure has at most **70 lines of logic**, excluding its signature, blank lines, comments, and ornamental delimiters. Multi-statement lines cannot be used to evade the rule. Split work by responsibility, not by arbitrary line count.

`odin run tools/check` uses tokens to distinguish Odin code from strings and comments. It includes procedure literals and multiline expressions. The current checker fails both soft-width findings and hard limits. Generated code must be split or redesigned to fit; it has no blanket exemption. Inherited Zen records remain an explicitly historical archive and are not presented as compliant new source.

Prefer small values, distinct IDs, exhaustive tag switches, checked arithmetic, explicit limits, and narrow package imports. Avoid unchecked pointer casts, disabling bounds checks, recursion driven by input, and thread-shared mutation. Errors due to user data return results. Assertions state internal invariants that have already been established by validation.

Memory safety

Guidedog must never crash on its input or on a failed allocation. Odin gives the tools; these rules say which ones every change uses. Tests enforce the rules that a scan can check (`tests/memory_safety_test.odin`); review enforces the rest.

- **Check every allocation.** `make`, `new`, `clone`, `aprintf` and the rest return an `mem.Allocator_Error` or an `ok`. A caller takes the value and the result together (`x, err := make([T], n)`), stops with `or_return`, or chooses a fall-back with `or_else`, and never indexes, dereferences, or passes on a value whose allocation failed. Work that cannot check each allocation (a library that goes on with what it has) allocates through a `host.Guard`: the guard remembers the first refusal and refuses every later allocation, so the work only shrinks, and the caller fails the whole step with `host.memory` when the work ends. Arenas (`host.Arena`, `runtime.Arena`, Jinja's own) stay consistent when their backing allocator refuses a block; `mem.Dynamic_Arena` does not, and is not used in product code (`tests/memory_safety_test.odin` checks). Code that cannot survive a refusal, such as Odin's parser, runs on memory the caller's allocator never refuses (odindoc parses in virtual memory of its own) within a bound reserved ahead. Tests refuse every allocation of a run in turn (`tests/allocation_failure_test.odin` and the libraries' `memory_test.odin` and `refusal_test.odin`).
- **Bounds checks stay on.** No `#no_bounds_check` and no `-no-bounds-check`, in code, build commands, scripts, or documentation. A loop that needs a check elided slices once before it, so the compiler proves the bound.
- **No pointer arithmetic or multi-pointers outside C bindings.** `[^]T`, `uintptr` arithmetic, and `transmute` between unrelated types belong to the

bindings of Graphviz, tree-sitter, the Typst bridge, and curl, which pass C’s memory to Odin as slices at their boundary. Elsewhere a region is carved by slicing the bytes first (bounds-checked) and then reinterpreting the slice (`slice.reinterpret`); the offset of a slice within its text comes from `core.offset_in`, which checks that it lies inside. Reading a frame’s address to measure the stack, or a native `#soa` slice’s header, is allowed with a comment saying so. Callback user data is the one `rawptr` cast allowed: the callee casts back to the type its registrant passed.

- **No views past their memory.** A string or slice into an arena, a session, or the temporary allocator is used only while that memory lives; what outlives a step is copied into the caller’s allocator before the step’s arena is freed (as `scratch_close` copies reports and problems). Library code never calls `free_all(context.temp_allocator)`.
- **Nil-checked pointers.** A procedure that finds something returns `^T` and `nil`, or `(T, bool)`, or `Maybe(T)`; its callers check before dereferencing. Tables that point into a struct use typed pointers (a union of them where kinds differ), never `rawptr`.
- **Ownership is explicit.** Each entry point owns an arena of its own, freed by one procedure (`unload`, `release`, `destroy_environment`) with `defer` at the call site.
- **AddressSanitizer.** `tools/sanitize.sh` runs every suite and the end-to-end tests with `-sanitize:address` before a release, and after changes to memory handling.

Write documentation as an explanation addressed to a careful colleague. Define a term before relying on it. State an invariant beside the algorithm that needs it. Give a small example, then show where the example stops applying. Explain why a choice exists and which observable behavior depends on it. Use plain English and precise claims, following the requested spirit of Feynman, Knuth, Lamport, and Dijkstra without imitating their prose or inventing quotations.

A public procedure’s comment states what it does, required preconditions, ownership, view lifetime, failure behavior, and any complexity bound that has been justified. A format record has a mapping table, conformance ledger, target limitations, policy refusals, and regression fixtures. “Unsupported” must identify a specific construct.

RST project and build workflow

The discussions are a separate Guidedog project. The manual is another project. Both begin with `guidedog quickstart`. A record’s prose belongs in `records/.template/rfc-template.rst` supplies new drafts. `registry.rst` lists identities and states. `bundle.rst` supplies the reading order. The CLI refreshes their managed regions without discarding surrounding prose.

```
guidedog gds index
guidedog gds check
guidedog gds build --target=both
guidedog gds build guidedoc-odin --target=pdf
guidedog build html docs/gds -W
```

RST publication uses `lib/project` and Guidedoc, like any other project. HTML is a coherent site in `docs/gds/_build/html`. A selector limits PDFs, not the

linked HTML graph. PDF builds produce one book per active record in `docs/gds/_build/pdf`. There is no combined design book by default. Typst is required for PDF, not for RST HTML.

The HTML template supplies navigation, search, and a page outline. The default is light. Readers may choose dark or system appearance. PDF uses the project book template. Graphviz diagrams render to static SVG assets. A typesetting template can remain Typst; the authored record is RST.

Legacy Typst catalogs remain supported by the library for existing callers. They use their original publication route. A single RST project must not mix native Typst records into its active record catalog.

Number assignment and validation

Guidedog has its own number sequence. GDS 0001 defines this process; the next assignment was 0002 for the CLI. GDS 0003 is the engine record. Records 0004 through 0006 cover budgets, the Odin domain, and projects. The next available number is 0007. Inherited zenfmt records do not reserve Guidedog numbers. New drafts retain `XXXXX` until a maintainer promotes them. Promotion chooses the next number after existing Guidedog records, explicit Guidedog reservations, and local `gds/NNNN` branches.

Validation checks unique assigned numbers and slugs, legal states, metadata and registry agreement, real source paths, and compilable active entry points. A record marked historical may link to the sibling repository but must not be included in the active PDF build unless its dependencies are present. Promotion updates the filename, metadata, registry, and bundle together. The current CLI does not rewrite prose links: the author reviews and repairs inbound references as part of promotion.

Guidedog CLI management contract

The implementation reference is `../docodin/cmd/docodin/dod.odin`, its DOD process, and its CLI reference. That implementation provides `new`, `list`, `index`, and `check`. It assigns permanent numbers at creation and includes an `ideation` state. Guidedog deliberately retains ZDS 0001's `XXXXX` drafts and six states instead; `promote` and `state` add explicit management of that inherited workflow.

```
guidedog gds new "Incremental project builds" --labels=architecture,build
guidedog gds list --state=prediscussion --label=build
guidedog gds show incremental-project-builds
guidedog gds promote incremental-project-builds --dry-run
guidedog gds promote incremental-project-builds
guidedog gds state 0002 --to=accepted --reason="Review converged"
guidedog gds index
guidedog gds check
guidedog gds build --record=0002 --target=pdf
```

All commands accept `--dir=DIR`, defaulting to `docs/gds`, and `--help`. Diagnostics use the standard Elm-style text or JSON mode and terminal color policy. `list` and `show` support `--format=table|json`; JSON contains metadata fields and source paths, never terminal decoration. Read-only commands never rewrite files.

Command	Required behavior
<code>new</code>	Create records/XXXXX-slug.rst from the local template. Set date, labels, title, author, and prediscussion. Refuse an existing slug.
<code>list</code>	Sort assigned records numerically, then drafts by slug. Filter by state and label; show historical records only with <code>--include-historical</code>.
<code>show</code>	Resolve number, exact slug, full title, or filename and print metadata and path. Ambiguous identifiers return candidate names without choosing one.
<code>promote</code>	Validate a draft, allocate the next unreserved number, rename it, set discussion, and update registry and bundle references.
<code>state</code>	Apply only a legal transition; require a reason, append review history, and update registry metadata. It never performs a Git commit or publication.
<code>index</code>	Rebuild derived registry entries and bundle includes deterministically; preserve surrounding prose and explicit GDS reservations. <code>--check</code> reports drift only.
<code>check</code>	Validate metadata, unique identities, paths, dates, labels, states, required sections and index/bundle agreement. Prose links need review.
<code>build</code>	Render selected records or all active records to HTML and individual PDFs. HTML builds refresh the complete catalog index; PDF builds create no index.
<code>recover</code>	With <code>--rollback</code> , restore journaled originals unless later edits conflict.

`new` accepts `--author`, `--slug`, and `--labels`. If author is absent, use configured Git identity; if absent, request `--author` with a concrete diagnostic. Do not invent an identity or change Git configuration. Slugs are normalized safe basenames; reject empty names, separators, and path traversal. Labels come from an explicit project catalog: architecture, parser, renderer, cli, build, theme, typst, library, documen-

tation, process, security, compatibility, performance, and release. Performance labels do not authorize premature optimization.

`promote` accepts an optional `--number=N` for maintainers and checks that it is unreserved in Guidedog records, explicit reservations, and local `gds/NNNN` branches when Git metadata is available. It never fetches remotely. Refuse numbers outside 0001 through 9999, duplicates, and previously assigned GDS numbers. `--discussion=URL` records a real discussion URL when supplied; otherwise leave its absence explicit. Number assignment is not design acceptance and does not automatically open a pull request.

`state` requires `--reason`; moving to `committed` also requires `--evidence` naming implementation and validation artifacts. This records a maintainer's decision, not an inference by the program that code is correct. No `--force` bypass of the state graph is provided. Supersession of a committed record uses a successor reference and leaves the historical state intact.

Mutation commands support `--dry-run` to show paths and metadata changes. They lock the GDS root, reread identities under the lock, stage changes, validate them, and commit with a recovery journal. Multiple file renames are not called atomic. A failed or interrupted operation must roll back or leave a journal that the next command reports with `guidedog gds recover --rollback`. Recovery verifies recorded original and intended contents and refuses to overwrite later edits. A live competing lock reports that the directory is busy. Kernel locks release on process exit; the persistent lock file is not itself a stale lock and is not removed.

The metadata parser recognizes a declarative RST field-list preamble; it must not execute arbitrary Typst to list or promote records. Unsupported dynamic metadata receives a direction to use literal fields. Rewrites touch that preamble and managed registry/bundle regions, preserving the record's prose. Escape strings as Typst literals. Mutating commands never overwrite an existing draft or output unless that operation's contract explicitly allows regeneration.

`check` is read-only. An optional `--render` additionally compiles active records; metadata checks must work without a Typst backend. Exit codes follow the general CLI contract: 0 success, 1 invalid records or drift, 2 usage, 3 resource limits, 4 host/backend failures, 5 internal errors, and 130 cancellation. `list` with no matches succeeds with an empty result. Diagnostics always identify the record, problem, consequence, and exact repair.

Acceptance tests cover new-record defaults, slug collisions, concurrent promotion, reserved numbers, legal and illegal transitions, idempotent indexing, malformed metadata, JSON output, dry-run with no writes, interrupted updates, and recovery that preserves later edits. Run them in temporary directories, never against the repository's actual discussion history. CLI management may use host allocations; that does not weaken Guidedoc's zero-allocation conversion boundary.

Security and operational considerations

Architectural decisions about parsing, resource limits, and publication must be traceable. Ambiguous status or invented implementation claims can cause unsafe assumptions in code. Review must separate implemented evidence from proposals and identify the owner of every resource boundary.

A record's metadata, examples, and links are reviewed along with its specification. Diagram dependencies must be pinned if external packages are used. Current diagrams use Graphviz directives. Guidedog renders vector output without a downloaded Typst package. `guidedog gds check --render` compiles the active records; `tools/check` checks implementation sources. Documentation prose is exempt from the source width and procedure limits.

Backwards compatibility and migration

GDS is the new name for the inherited ZDS process in this project. Preserve existing numbers and historical meaning. Rename active metadata and paths from `zds` to `gds`. Keep original copied Zen artifacts in the archive where needed. GDS 0001 is the adaptation of ZDS 0001, not a replacement lifecycle invented for this repository.

The old architecture remains historical. New Odin proposals do not silently amend `zenfmt`'s implementation. No chapter directory forms part of the GDS structure.

Alternatives considered

A book would make individual proposals harder to discuss and assign stable status. A new PEP-style state machine would conflict with the existing ZDS process. Automatic numbering at draft creation would lose the local placeholder workflow. These options are not adopted. A future manual may still explain accepted behavior separately.

Acceptance criteria and unresolved questions

The implemented process preserves the inherited six states and permanent numbering. The CLI checks metadata and required sections, regenerates the registry and reading order, and uses locked, recoverable transactions for promotion and state changes. Integration tests cover legal and illegal transitions, reserved numbers, dry runs, concurrent edits, interrupted journals, and recovery that preserves later edits. The actual RST template is exercised by `tests/manual_test.odin`. All six active records are built separately as HTML and PDF during this review. The external discussion venue remains unassigned; no external approval is claimed. GDS 0002 retains the broader CLI obligations not completed for the beta.

Review history

29 September 2026: rename Zen Discussions to Guidedog Discussions; preserve the lifecycle; replace unavailable Zig build instructions with actual Typst commands; incorporate RFC, Oxide RFD, and PEP structural guidance; remove chapter-based drafts.

29 September 2026, tooling review: replace temporary direct-build instructions with the Odin CLI. Record HTML catalog and individual PDF behavior, independent Guidedog numbering, snapshot-based recovery, and manual review of prose references. Keep the existing discussion state; no lifecycle acceptance is inferred from tests.

30 September 2026: add the memory-safety rules to the coding rules: checked allocations, `host.Guard` for work that does not stop at a failed allocation, bounds

checks always on, no pointer arithmetic outside C bindings, no views past their memory, nil-checked pointers, and AddressSanitizer runs.

References

- Docodin `cmd/docodin/dod.odin`, `docs/dod/0001-dod-process.rst`, and CLI reference.
- Original ZDS 0001, preserved at `legacy/0001-zds-process.rst` and in `../zenfmt`.
- RFC 7322: RFC Style Guide.
- RFC 2119 and RFC 8174: requirement keywords.
- Oxide RFD 1: Requests for Discussion.
- PEP 1: PEP Purpose and Guidelines.
- PEP 12: Sample PEP Template.

Source-format revision — 2026-10-01

The authored records, catalog, and creation template are now RST. The CLI retains the six-state lifecycle and recoverable metadata transactions. Publication uses the project engine. This is an editorial and toolchain revision; it does not change this record's review state or assert acceptance.

Lifecycle event

2026-10-11: discussion → published. Maintainer-requested reconciliation: the RST workflow and inherited lifecycle are implemented and reviewed.

Lifecycle event

2026-10-11: published → committed. RST discussion management and recoverable lifecycle transactions implemented. `tests/manual_test.odin`; workflow, concurrency, recovery and date tests; native GDS strict HTML/PDF verification; `docs/manual/evidence/beta-review-20261011.md`.

Number: 0002
Title: Guidedog: CLI and documentation toolchain
State: published
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-29
Updated: 2026-10-11
Discussion: Local review; no external discussion URL assigned
Labels: cli

Guidedog: CLI and documentation toolchain

Abstract

This record specifies the Guidedog command line: discussion management, single-document conversion, and the documentation project builder. All three are implemented. The executable converts reStructuredText, CommonMark, and native Typst to HTML and PDF, builds a responsive website and a PDF book from a project, and embeds the Typst 0.15.1 compiler when built with its bridge. Typst

packages and system fonts are supported. Open items, such as worker isolation for hostile Typst and a cross-document symbol index, are listed at the end.

Motivation and scope

Users need one command that makes output predictably and explains failures in terms they can act on. The CLI should share engine semantics with embedded callers, while owning terminal behavior, files, resources, Typst, and publication. This record separates what is implemented from what remains open; an implemented command does not by itself establish that every acceptance gate below is met.

Implemented specification: discussion CLI

`lib/cli` owns arguments, validation, terminal presentation, and dispatch. `lib/gds` owns record metadata, catalog selection, lifecycle plans, and recovery. RST publication delegates to `lib/project`. Guidedoc reads and renders the prose. Typst sets PDF pages. RST HTML does not require a Typst installation.

```
guidedog gds new "Cache policy" --author="Your name" --labels=build
guidedog gds list --state=prediscussion
guidedog gds show cache-policy --format=json
guidedog gds promote cache-policy --dry-run
guidedog gds index
guidedog gds check
guidedog gds build --target=both
guidedog gds build guidedoc-odin --target=pdf
```

A selector is a number, exact slug, full title, or source filename. Ambiguous selectors return candidates. `--dir` selects the discussion project. The default remains `docs/gds`.

Metadata is an initial RST field list. Scalar values are plain text. Authors and labels are JSON string arrays. Listing and editing never execute code. The local RST template supplies new drafts. An author comes from `--author` or Git's configured name. Missing authors receive a correction direction.

Mutations are separate from builds. A mutation holds the kernel lock while it plans and applies changes. The recovery journal records original and intended text. Rollback refuses to overwrite an edit made after the interrupted operation. Closing the lock handle releases the lock. A leftover lock filename is not a busy lock.

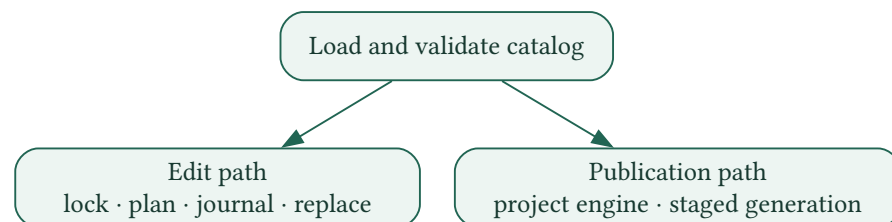


Fig. 3 Metadata edits and publication use different transactions.

HTML builds publish a complete site in the discussion project's `_build/html`. The catalog, search, labels, and cross-document links use the normal project graph.

A selected build still refreshes that coherent site. Selected PDFs use a separate generation in `_build/selected/<artifact>/pdf`; they do not replace the complete catalog in `_build/pdf`. PDF builds create one book per selected active record in `_build/pdf`. No selector means all active records. There is no combined PDF by default. A normal `guidedog build html docs/gds` or `build pdf` works too.

The project engine owns cache validity and generation publication. Compiler or renderer failure preserves the previous target generation. Each HTML or PDF target commits independently; `both` is not a transaction covering both target directories.

Source and metadata limits remain distinct from compiler costs. A record is limited to 1 MiB. Directory entries are limited to 4,096. The recovery journal is limited to 32 MiB. Documentation prose is exempt from source formatting limits. The external Typst compiler's allocations are not bounded by the host's working-memory budget.

Legacy Typst catalogs keep their old metadata and publication contracts. This RST project has no need for Typst's experimental HTML exporter. The source-format change does not promote this record to an accepted state.

Specification: conversion command-line behavior

The first executable is named `guidedog`; `convert` exposes Guidedoc before the project builder exists. This avoids shipping a second CLI name that later needs renaming. `convert` reports each reader's conformance through `formats`: CommonMark passes all 652 examples of 0.31.2; the `reStructuredText` reader is partial, with its gaps listed in the readers' coverage ledgers.

```
guidedog --help
guidedog --version
guidedog formats
guidedog convert guide.rst --to html --output guide.html
guidedog convert guide.md --to pdf --output guide.pdf
guidedog convert guide.typ --to pdf --output guide.pdf
guidedog convert - --from commonmark --to html --output -
```

`.md` selects CommonMark. `.rst` selects `reStructuredText`. `.typ` selects native Typst. An explicit `--from` overrides suffix selection; ambiguous or missing selection produces a direction to specify it. No heuristic claims that arbitrary plain text is reliably distinguishable as RST or Markdown. `--to` accepts only `html` or `pdf`; if omitted, the output suffix may select the target. Intermediate Typst is available through `--emit-typst path`, not as a third product target.

`formats` reports actual adapter availability, conformance status, and toolchain version. It must distinguish planned, partial, and conformant readers. Global flags include `--color=auto|always|never`, `--diagnostics=text|json`, and `--quiet`. Use `--` to terminate option parsing. Never evaluate a shell command from `argv`.

ANSI belongs at the terminal boundary

Use the installed Odin package `core:terminal/ansi`. Its constants are escape-code components, so a colored heading combines `ansi.CSI`, a color, and `ansi.SGR`. The terminal renderer is the only package allowed to emit these sequences.

```
import "core:terminal/ansi"

error_style :: ansi.CSI + ansi.FG_RED + ansi.SGR
```

```
reset_style :: ansi.CSI + ansi.RESET + ansi.SGR
```

Automatic color requires a terminal on the diagnostics stream, no `NO_COLOR`, and a terminal other than `dumb`. Explicit `--color` takes precedence. JSON and artifact bytes never contain ANSI styling. Diagnostic source excerpts escape terminal control characters, including embedded escape sequences from filenames or source text. Honor redirected stderr, narrow terminals, tabs, combining marks, and Unicode. Color supplements words and markers; it never carries meaning by itself.

stdout contains requested artifact bytes or machine-readable command output. stderr contains diagnostics. In JSON mode stderr is a versioned JSON-lines report stream. Help and version use stdout on success. Version output includes compiler, Guidedoc schema, and Typst backend versions when applicable.

Exit	Meaning
0	Completed successfully; warnings may exist
1	Invalid document, unresolved references, or unsupported target content
2	Command syntax or option error
3	Capacity or configured resource limit exceeded
4	Host I/O or Typst backend failure
5	Internal invariant failure; request a reproducible report
130	Cancelled by the user

Elm-style diagnostics are structured data

Every warning and error answers four questions: what happened, where, what this means for the result, and what to do next. Prefer a specific edit or an exact flag to advice such as “check your input.” The library returns facts; renderers choose prose layout, color, and machine-readable encoding.

A `Diagnostic` holds stable namespaced code, severity, source ID and byte span, problem kind, bounded typed arguments, consequence, and one or more directions. Directions contain a suggested replacement, a complete argv vector, or a named API field and value. Source text remains borrowed. Fixed inline numeric fields and static message templates allow capacity errors to be reported when report storage is itself full. `Result.primary` is always available by value on failure.

```
UNRESOLVED REFERENCE                                rst.reference.unresolved

guide.rst:18: See installation_ before continuing.
               ^^^^^^^^^^^^^^

I could not find a target named "installation".
The HTML page has not been published.

Add a target beside the installation section:

.. _installation:

Or change this reference to an existing target name.
```

Rendering calculates line and display column from byte offsets and a bounded line index. A report without source context uses an argv span, resource name, or workspace region. Reports never invent a source position. JSON retains the same code, evidence, consequence, and directions as the text view.

When report storage fills, retain the first failure in `primary`, count suppressed reports, and set an explicit truncation flag. In strict reporting mode, truncation fails the conversion. A bounded excerpt avoids copying a huge source line into the report. The host renderer may use ordinary allocations; a bounded-buffer renderer must remain available for allocation-free embedders.

CLI resource policy

Superseded in part by the draft “Memory budgets and what a build may read”: how a build’s budget is reserved and shared (one budget per build, `--budget`), what it counts and what it names as outside it, and which files a build, a graph, and Typst may read, trusted and under `--untrusted`. The limits below stand.

Initial defaults are proposed, not measured performance claims: 32 MiB per source, 256 MiB total prepared Odin storage, 256 nesting levels, 1,024 stored reports, 1,000,000 AST nodes, and 128 MiB output. Core receives limits explicitly; the CLI selects them. A separate project budget bounds aggregate files and resources. Each exhausted limit names the corresponding flag or API field and a concrete next value when computable. Byte and node capacities are checked independently.

A cancellation probe is checked at stage boundaries and at bounded work intervals. Compressed archives, network retrieval, and office formats are outside this release. Local paths are confined to approved roots with symlink-aware checks. Include and substitution expansion have depth, total-byte, and work limits.

Guidedog: the project builder

NOTE

The draft record “Guidedog: everything Sphinx does, the Odin way” (`0006-sphinx-replacement.rst`) replaces this section’s project builder: the `guidedog.toml` layout, `init`, and `build --target`. That record is the current statement of the project layout, `conf.toml`, builders, incremental builds, and publication. What this section says about the build cache, publication, confinement, and the themes still holds and is kept for its history; where the two differ, the draft governs.

The full application adds project discovery, document ordering, cross-references, templates, incremental builds, local preview, and publication. It reuses Guidedoc for RST and CommonMark and the Typst adapter for native Typst content.

```
guidedog quickstart docs
guidedog build html docs
guidedog build pdf docs
guidedog serve docs --port 8000
```

These commands are implemented in `lib/project` and `lib/cli/project.odin`; the superseded `guidedog init manual --template atlas` and `guidedog build --target html` forms no longer exist.

Configuration is declarative TOML: source roots, entry document, navigation, targets, language, theme tokens, allowed resources, and build budgets. It is not executable Python. Built-in assets and templates are embedded into the executable. User overrides are explicit files in the project.

Project graph and stable names

Discovery sorts normalized project-relative paths. Parsing collects document IDs, heading targets, named RST targets, citations, includes, and resource dependencies. Resolution constructs a deterministic symbol index and resolves cross-file edges before rendering. Duplicate IDs and cycles are diagnosed with both source sites. URLs are stable project IDs plus anchors, never ephemeral node indices.

For RST and CommonMark, the project host supplies read-only symbol and resource views. Conversion does not open neighboring files. Mixed-source books generate Typst chapter modules for RST and CommonMark and import native Typst chapter modules through a documented content-export convention. An arbitrary standalone `.typ` file can always be built alone; joining it into a book may require adapting page rules and exports. Never promise automatic merging of unrelated Typst programs.

Native Typst chapters declare navigation and exported cross-reference labels in a small project metadata contract. The adapter resolves these against the shared project manifest. It does not guess arbitrary runtime labels from source tokens.

Implemented today: discovery, duplicate-page diagnostics, a cross-file symbol index of labels, terms, domain objects, and documents, and a link pass that resolves references through it: to page URLs for the site, to chapter labels for the book, and to `#<document key>-<anchor>` in `singlehtml`. Native `.typ` documents become site pages through Typst’s HTML export and book chapters through `#include`. The native Typst metadata contract above is not implemented.

A build-cache key includes source and transitive dependency digests, reader version, configuration, renderer version, template, fonts, and Typst version; the implementation also includes the executable’s size and modification time, so a rebuilt `guidedog` never reuses an older conversion. A page’s key also covers what its references resolved to: the link pass records every lookup it makes, and the next build repeats them. Diagnostics are cached with the output they came from and reported again when it is kept. Development caches are host-owned and may allocate. A clean build and a cached build must emit the same artifacts for the same inputs, excluding explicitly requested timestamps.

Responsive HTML and the Typst book

The first HTML theme, proposed name **Atlas**, has a restrained neutral palette, a clear accent color, a readable article measure, and generous spacing. Desktop layout includes project navigation, article, and an on-page outline. On narrow screens, navigation becomes an accessible drawer and the article remains readable at 320 CSS pixels. Wide tables and code scroll within their containers.

Require semantic landmarks, skip links, keyboard navigation, visible focus, proper heading order, image alternatives, high contrast, and reduced-motion support. The page remains readable without JavaScript. Search uses a generated local index; no remote service is required. CSS, fonts, and icons ship with the binary, with license notices. Native Typst HTML is integrated through its supported export and template mechanisms; do not concatenate entire HTML documents into a page shell.

The PDF theme, proposed name **Folio**, is a Typst book template: title page, contents, numbered chapters, running headings, readable body type, restrained code styling, figures, tables, notes, citations, and useful page breaks. Test A4 and Letter, long URLs, wide tables, multilingual text, and embedded fonts. User content is passed as content and data, with a small documented theme-token surface.

Atlas and Folio are implemented and embedded. Desktop and phone-width pages and PDF pages were reviewed as rendered images; a formal accessibility audit remains open. A project `pdf.preamble` file adds package imports and show rules to the book.

Typst and the single executable

Typst's published compiler library is Rust. The selected design is a small Rust adapter exposing a versioned C ABI, statically linked into the Odin application. The conversion engine remains Odin. Release builds must prove that the linked binary needs neither an installed `typst` command nor a downloaded package or font. This is an integration decision to validate, not an existing upstream C API.

The ABI accepts pointer-and-length source and resource records, explicit target, limits, and diagnostic callbacks. Rust owns its compiler state and output until an explicit release call; no Rust object layout crosses the ABI. Copying into caller buffers is explicit. Convert panics into failures at the boundary where supported; never unwind through Odin frames. Allocation failure and hard process termination cannot be promised recoverable merely by catching panics.

The adapter implements Typst's resource environment. Project files are read only under the Typst root. **Typst packages are supported**, because publication-quality PDF depends on them and Guidedog writes Typst rather than LaTeX: `@preview` and `@local` imports resolve like the typst CLI, from the package data directory, then the cache, then a Typst Universe download into the cache. A package reads only inside its own directory and has no shell or network access of its own; that is Typst's sandbox, and it makes a package no riskier than a LaTeX package. Versions are pinned by the import itself (`@preview/name:1.2.3`). `--offline` (or `pdf.packages = "offline"`) restricts builds to packages already on disk, for CI and air-gapped machines.

Fonts also follow the typst CLI: extra `--font-path` directories first, then system fonts, then the fonts embedded in the executable as a fallback, so a book can use the fonts installed on the machine. `--embedded-fonts-only` (or `pdf.fonts = "embedded"`) ignores system fonts for byte-reproducible output across machines. Trusted templates are embedded. No other filesystem or network access is delegated to document content. For hard time or memory isolation, a future implementation would invoke an internal worker and apply OS limits. This worker mode is not implemented. Per-platform containment and cancellation

must be verified before hostile native Typst is advertised as safely isolated. An in-process compiler is not a sandbox.

The bridge exists in `native/typst_bridge`: ABI version 3, one request carrying root, main file or generated text, target, date, package mode, cache, font mode, and font paths; one owned result released by `gd_typst_free`; panics become an internal-failure status. It builds offline from the cached crates on macOS; Linux vendors OpenSSL statically for the Typst bridge. The complete Unix CLI still links system libcurl. Release builds pass `-define:GUIDEDOG_TYPS_TBRIDGE=true`; the default build keeps the development backend.

A development adapter may invoke an installed, pinned Typst CLI with an `argv` array. It is labeled a development backend and does not satisfy the single-binary release gate. Static-link feasibility, binary size, licensing, supported platforms, HTML export fidelity, and worker containment are explicit early technical checkpoints.

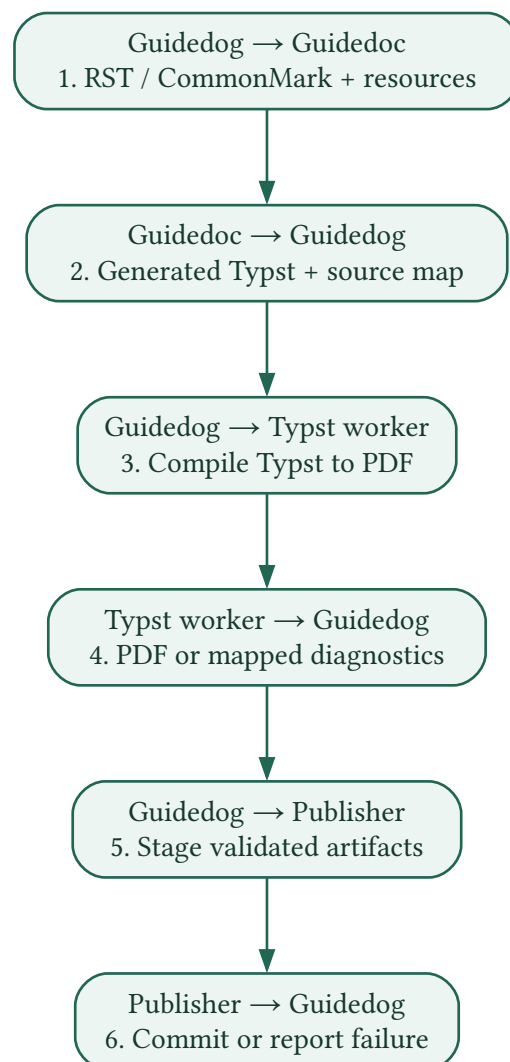


Fig. 4 Native .typ input starts at step 3. Failed compilation never reaches step 5.

Publication is a host transaction

Conversion to a path writes a sibling temporary file, flushes and closes it, then replaces the destination using the platform's supported rename operation. Refuse existing output unless `--force` is explicit. stdout output cannot be rolled back; finish conversion into staging memory before emitting it.

A website or book with resources is a set of files, so two individual renames do not form an atomic transaction. Stage an entire versioned build directory and switch an application-owned current-version pointer where supported. On platforms without an atomic switch, preserve the previous build and document recovery. An optional provenance manifest records versions, digests, diagnostics, and resources; it belongs to the same staged set. Crash durability requires directory sync behavior where available and is tested separately from logical all-or-nothing publication.

Superseded for project builds by the Publication section of 0006-sphinx-replacement, which builds the whole next output folder beside the live one and switches the two (in one step where the system can), journals the switch so a failure is undone and a crash finished by the next build, publishes the environment with the output, and decides `-w` before publishing. The decision below, whose file-by-file moves could not survive a file that becomes a folder, is kept as history.

Implemented decision. A project build stages each file it writes in a folder beside the output folder, and a file whose bytes are already published is only recorded. When every unit of the build succeeded, the staged files are renamed into place, the files only the previous generation made are removed, and the manifest is saved last. A failed or interrupted build removes the staging folder; the published output is untouched. The switch is file by file rather than one pointer: the output folder stays the path users serve and preview, and unchanged files keep their times. A crash while files are being moved leaves the old manifest, so the next build rewrites exactly the pages that differ.

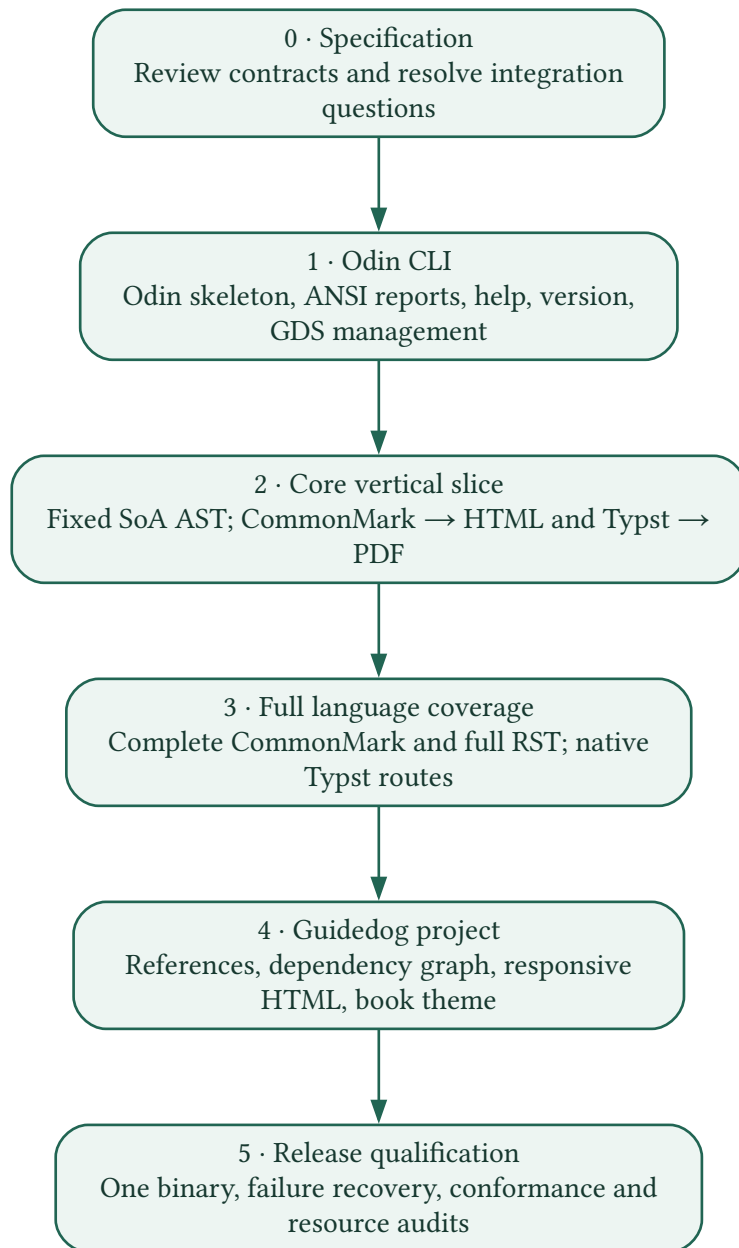
Delivery phases

Fig. 5 Each phase proves behavior before adding a larger surface. No phase promises speed.

Phase 2 first uses a development Typst backend while a separate feasibility check validates static linkage. Its CommonMark slice is explicitly partial until the full suite passes. Phase 3 cannot close with a partial RST parser marketed as complete. The release gate requires all declared routes to work from a clean machine without Python or a Typst installation, and without a network download unless a document imports a Typst package that is not cached. Optimization follows working behavior and measured evidence, not an earlier calendar milestone.

Status on 29 September 2026: phases 1 and 2 are complete. Phase 3 is complete for CommonMark and native Typst and partial for RST, as its ledgers record. Phase 4 is implemented except the native Typst chapter contract. Phase 5’s single binary works on macOS and Linux; Windows runtime, worker isolation, and font-license review remain.

Security considerations

Source content, terminal excerpts, include paths, and generated Typst are distinct trust boundaries. The rules above require escaping, approved resource roots, explicit raw-content policy, and isolated Typst execution when hard bounds matter. No document may acquire filesystem or network authority by naming a resource. The single-binary packaging requirement is not itself a sandbox guarantee.

Backwards compatibility

The CLI, conversion, and project commands described above are available in this checkout. The diagnostics JSON schema is version 1; changes to it require a new schema version. The project does not emulate every Sphinx extension or configuration script. Full RST support and Sphinx-specific extension support are separate claims. Existing source files must receive useful diagnostics when policy prevents a feature.

Alternatives considered

A separate converter executable would duplicate the future Guidedog command surface. An installed Typst command is useful during development but fails the single-binary release requirement. Reimplementing Typst in Odin would replace a mature compiler with an unrelated language implementation. A narrow static bridge preserves the Odin application and reuses the real compiler. Mandatory sidecars for every byte conversion would improperly move host policy into the core.

Discussion management

`guidedog gds` is part of the first CLI milestone, following docodin’s integrated DOD commands and the inherited GDS process. The CLI implements `new`, `promote`, `list`, `show`, `state`, `index`, `check`, `build`, and `recover` according to GDS 0001. GDS HTML/PDF builds use the embedded Typst compiler when it is linked, and otherwise the development backend. Both this repository and documentation projects can select a GDS root with `--dir`.

Acceptance criteria and reference implementation

The CLI must pass help, usage, option precedence, redirection, JSON, ANSI, exit-code, and report-overflow tests. The project builder must pass the engine discussion’s conformance and resource tests, the publication failure tests above, and visual review of HTML and PDF templates. Single-binary qualification requires a clean machine with no Python or installed Typst.

Evidence

The reference implementation is `cmd/guidedog` with `lib/cli`, `lib/gds`, `lib/host`, `lib/typst`, `lib/project`, `lib/defaults`, the Guidedoc packages of GDS 0003, and `native/typst_bridge`. On 29 September 2026:

- `tests/` holds 137 end-to-end tests (30 September 2026), among them: discussion workflows, recovery, dates, Typst builds; conversion to HTML and PDF from RST, CommonMark, and Typst; overwrite refusal; JSON and text diagnostics; include confinement; offline package resolution from a private cache; project init, build, incremental rebuild, failed-build preservation, and configuration errors; the preview server; long headings; standard-input includes; exact and one-short capacity for every storage region; memory budgets; publication of whole generations; diagnostics that do not depend on earlier builds; dependencies on tags, nested templates, and moved objects; `singlehtml` addressing; link confinement; and `--untrusted`.
- Package suites (test procedures): core 18, host 5, project 18, i18n 28, gettext 31, jinja 46, highlight 64, graphdog 22, inventory 4, fetch 1, CommonMark 22 (all 652 examples), RST blocks 208, RST inline and resolution 65, Sphinx 73, HTML 28, Typst 24, text 13. Each includes a zero-allocation test where the package is inside the core boundary.
- Every suite passes on macOS arm64 and Linux x86_64, with the installed and the embedded Typst backends. The embedded executable runs with no `typst` on PATH: 64 MB on macOS, 80 MB on Linux. The earlier C-runtime-only claim was incorrect: system `libcurl`, the C++ runtime, and their transitive dependencies remain. Windows type-checks with and without the bridge.
- Ctrl+C during a PDF build exits 130, leaves no staged file, and publishes nothing.
- `tools/check` passes every source file. A 460 KB, 4,000-section Markdown file converts to HTML in 0.07 s at 33 MB peak memory. Each document and page is converted in its own arena, so a 1,500-document site builds at 15 MB peak. These are observations, not performance claims; every page carries the full navigation, so site size grows with the square of the page count.
- An independent review of the host packages found eleven defects, from anchor sizing to memory growth, output validation, and terminal escaping; all are fixed with tests.

```
odin build cmd/guidedog -out:build/guidedog
cargo build --release --manifest-path native/typst_bridge/Cargo.toml
odin build cmd/guidedog -define:GUIDEDOG_TYPST_BRIDGE=true -out:build/guidedog
odin run tools/check
odin test tests -out:build/guidedog-tests
```

This record is retained as `published`, with incomplete obligations below. GDS 0004 specifies the implemented resource policy; GDS 0006 supersedes the original project surface and publication design. The remaining obligations prevent marking this broader record `committed`.

Decisions still requiring implementation evidence

Hard time and memory isolation for hostile native Typst still needs a worker mode and per-platform verification. The native Typst chapter metadata contract is not implemented; Typst is supported by standalone conversion and PDF

templates. Windows runtime validation is still outstanding. Embedded font licenses are recorded with the fonts; they are no longer an open documentation item. Packaging must account for actual runtime dependencies: embedding Typst and static Graphviz does not remove the system `libcurl` link on Unix. A clean machine distribution check remains a separate release gate. The RST ledgers no longer have the previously listed syntax gaps; their pinned tests and conservative reader descriptor state the evidence accurately.

These limits do not prevent a beta for trusted documentation projects on macOS and Linux with the declared dependencies. They do prevent advertising a Windows production release, hostile-input isolation, or a dependency-free Linux executable.

Review history

29 September 2026: separates terminal and host policy from the allocation-free engine; specifies full source-language scope, Typst PDF routes, and later project features. The initial revision preceded the implementation.

29 September 2026, implementation review: record the working GDS CLI, HTML catalog, individual PDFs, journal recovery, local dates, and portable host boundaries. Correct the inherited assignment from 0017 to Guidedog's 0002; zenfmt numbering does not reserve GDS numbers. Record the 18-test evidence and unresolved acceptance gates. Preserve discussion status because the full record is not yet implemented.

29 September 2026, implementation of conversion and projects: record `convert`, `init`, `build`, and `serve`; the embedded Typst bridge; Typst packages and system fonts at the user's direction, replacing offline-only packages and embedded-only fonts; Ctrl+C handling; Linux runtime evidence; and the remaining open items.

29 September 2026, review of the build pipeline: record whole-generation publication with a manifest, recorded lookups and replayed diagnostics, session memory budgets reserved before allocation and shared by reading threads, confinement of linked files, `--untrusted`, and the current test evidence. The cross-file symbol index is implemented.

References

- GDS 0003, Guidedoc: Odin conversion engine.
- GDS 0001, The Guidedog Discussion Process.
- [Odin ANSI package](#).
- [Typst compiler crate 0.15.1](#).
- [Typst HTML export](#).

Lifecycle event

2026-09-29: prediscussion → discussion. Assigned a permanent number and opened for discussion.

Source-format revision — 2026-10-01

RST discussions now use the normal project engine. The design record remains in discussion. See GDS 0001 for the revised authoring contract.

Lifecycle event

2026-10-11: discussion → published. Retain the implemented CLI contract with explicit unfinished worker, native Typst chapter, Windows and packaging obligations.

Number: 0003
Title: Guidedoc: Odin conversion engine
State: committed
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-29
Updated: 2026-10-11
Discussion: Local review; no external discussion URL assigned
Labels: architecture

Guidedoc: Odin conversion engine**Abstract**

This proposal defines Guidedoc, a reusable Odin library with a zero-allocation core, a native multi-SoA AST, and direct HTML and Typst renderers. It reads reStructuredText, implemented independently from the Docutils specification, and CommonMark 0.31.2. Native Typst uses an explicit toolchain adapter, and PDF is always compiled from Typst. There is no writer-lowering layer and no speed claim. The companion Guidedoc CLI discussion, GDS 0002, defines host services and publication. The engine is implemented; the evidence and open items are at the end.

Motivation

The existing zenfmt design supplies useful document and ownership ideas, but its Zig APIs and allocator strategy do not express the intended Odin library contract. docodin, an Odin engine that reproduces Docutils byte for byte, supplies measured research on handles, struct-of-arrays storage, and source spans. Guidedoc has a different goal: its own semantic HTML and Typst output from the language specification. This proposal specifies that contract and separates the allocation-free engine from resource acquisition and the Typst runtime.

Purpose and boundaries

A universal converter needs a common language for documents. It must retain meaning until the chosen target forces a decision. A heading remains a heading; a table remains a table; an unsupported feature becomes an explicit loss or a refusal. A successful conversion must never conceal that distinction.

“Universal” describes an extensible architecture, not a promise that every format is implemented or that every pair can preserve all information. The supported inputs are reStructuredText, CommonMark, and Typst markup. The output targets are HTML and PDF. PDF always goes through Typst: reStructuredText and CommonMark produce Typst source; native Typst input retains its own semantics. Office formats and Markdown output are outside this design.

Three responsibilities

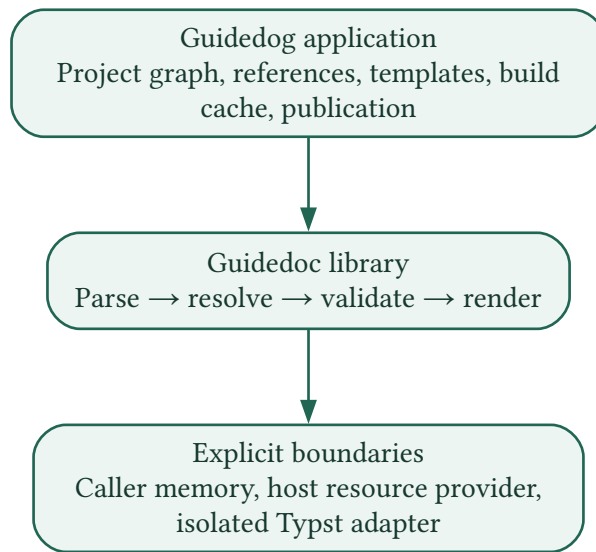


Fig. 6 Responsibility flows downward. Core conversion does not import the application.

The Odin core accepts reStructuredText or CommonMark bytes and returns HTML or intermediate Typst bytes with structured diagnostics. A separate Typst adapter compiles generated or native Typst source into HTML or PDF as appropriate. The core does not open paths, print messages, look at environment variables, fetch URLs, or discover plugins. The application decides which files may be read and which artifacts may be published. An editor, a server, and a command line client therefore share the same conversion semantics.

The host may allocate while acquiring input or preparing a workspace. Guidedoc’s conversion procedures may not allocate. The Typst compiler has its own memory management; the zero-allocation guarantee does not extend into that dependency.

What we retain, and from where

The sibling repositories were inspected at `../zenfmt` and `../docodin`. zenfmt’s ZDS 0013 supersedes the tree and lowering sections of its ZDS 0002; its `core/src/host.zig`, `options.zig` and `report.zig` also inform this proposal. docodin’s DOD 4 (data-oriented storage), DOD 12 (parser research) and DOD 13 (source spans) record measured Odin experience. These are design inputs, not ported code.

Inherited idea	Odin decision
Flat semantic tree (zenfmt)	Typed IDs and native SoA slices over fixed storage.
Handles, not pointers (docodin)	distinct u32 node IDs; bounds checks stay on.
Hot columns, cold side tables	Kind and subtree end are scanned; payloads are cold.

Inherited idea	Odin decision
Byte spans and line index	Spans are byte ranges; lines are computed on demand.
Writer lowering (zenfmt)	Omitted. Direct HTML and Typst renderers.
Docutils output fidelity (docodin)	Not a goal. The specification is the baseline.
Elm-style reports	Structured evidence and concrete next actions.
Compile-time bundles	Ordinary imported descriptors in explicit slice registries.
Arena growth	Caller storage with checked capacity; core never grows an arena.
Mandatory adjacent manifest	Opt-in host preservation artifact.

We do not translate Zig’s comptime type factories or generated routing matrix. Odin’s package imports and procedure values are enough to connect a reader and a writer. A small indirect call per stage is acceptable; dispatch per character is not.

Terminology

A **workspace** is borrowed mutable storage for one active conversion. A **document** is a validated immutable view of a workspace snapshot. A **reader** constructs that view. A **pass** transforms it. A **renderer** encodes HTML or intermediate Typst markup. An **attribute** is optional keyed information attached to a node; it plays the role zenfmt called a facet. A **resource** is an included source, image, or other byte asset whose acquisition belongs to the host.

A **render check** finds unsupported target features before emission. It does not search for alternative representations. An **artifact** is a resulting byte stream. **Publication** means making a staged artifact visible at its destination. These terms distinguish conversion success from filesystem success.

Non-negotiable contracts

- No allocator calls inside core initialization, parsing, resolution, validation, rewriting, render checking, diagnostic construction, or bounded-buffer writing.
- No mutable global state. A registry may be shared when immutable; workspaces may not be shared by concurrent conversions.
- Every input-dependent loop has a bound or a progress invariant. Nesting has a checked depth limit, `Limits.max_depth`, whose excess is a `Limit` result. The builder, validation, the HTML renderer, and the CommonMark reader keep their nesting in caller-provided stacks; see “Recursion is bounded by the depth limit and the stack budget” for the RST reader, the C++ declaration parser, and the other renderers.
- Success returns valid output. Failure returns a stable code and a useful action. Corrupt input is a recoverable result, never an assertion or a panic. Content that does not fit a bounded buffer is a reported failure, never silently cut short.
- Declared losses are available before publication. Strictness can refuse them.

- Registry validation rejects duplicate format IDs; nothing is silently replaced. `convert` validates its registry on every call, so an unchecked registry cannot convert with the first of two adapters.
- Source maps use byte offsets into named UTF-8 sources. Display columns are a rendering concern, not a second coordinate system inside the engine.

Memory is part of the API

“Zero allocator” means that Guidedoc borrows already available storage and never asks an allocator for more. It does not mean zero memory, zero copying, or an unbounded document in a fixed buffer. Advancing a used-count inside a caller-owned slice is allowed. Calling an allocator, including `context.temp_allocator` or an arena allocator over a caller buffer, is not allowed inside the guaranteed boundary.

The host chooses stack arrays for small work, a fixed pool for a server, or one allocation per region for a CLI session. Large automatic arrays are unsuitable for small thread stacks. Documentation shows this choice without hiding it in a convenience constructor.

Storage regions and lifetime

Region	Contents	Valid until
Input	Borrowed UTF-8 bytes and names, main and included	Last span or report use
Snapshot A / B	Node columns, payload tables, attributes, text pool	Reset of that bank
Scratch	Reader skeletons, stacks, symbol slots, render state	End of the stage
Reports	Fixed diagnostic records	Next conversion or reset
Output	Caller-owned byte buffer and used count	Caller reuses the buffer

All regions must be disjoint, suitably aligned for their element types, and alive for the advertised lifetime. Typed slices avoid requiring callers to align raw bytes for the snapshot; scratch is the one raw byte region. Initialization checks lengths and rejects overlapping regions. Neither strings nor slices convey ownership in Odin, so borrowing rules appear in the procedure documentation.

Node text is always owned by the snapshot’s text pool. Readers copy text into the pool after escape processing and entity decoding. The input is referenced only through source spans, for diagnostics and source maps. This costs one linear copy and removes a whole class of lifetime errors: a snapshot never aliases input bytes or the other bank, so validation checks one pool and a rebuild copies nothing it cannot own.

Scratch is a marked bump region over a caller byte slice. A stage takes typed, aligned slices with `scratch_take`, records a mark, and releases to it when finished. Taking scratch advances a used count inside the caller’s slice; it is not an allocator call. Exhaustion reports the scratch region and the minimum additional bytes.

Two snapshot banks permit rebuild passes. A pass reads A and writes B, validates B, then swaps the active bank. On failure, A remains valid. An unchanged pass returns A immediately. Bank B may have zero capacity when a caller runs no passes; a pass then fails before it runs with `Capacity` and a need whose region is `Second_Bank` (`core.storage.second_bank`), which names the cause rather than the nodes region. There is no persistent sharing or copy-on-write mechanism in version one.

Sizing and carving cannot disagree about bank B. The one-buffer form, `workspace_size(n)` and `init_workspace_from_bytes(ws, slab)`, always plans one bank and takes no bank argument. A workspace for passes is sized and carved from one value: `plan := slab_plan(n, second_bank = true)`, `plan_bytes(plan)`, and `init_workspace_with_plan(ws, slab, plan)`. (An earlier draft gave both one-buffer procedures a `second_bank` argument with a default; sizing with it and initializing without it left bank B empty, and the failure surfaced only at pass time as a need for one node.) `init_workspace_with_plan` is the authoritative path, which the one-buffer form only chooses a plan for. All sizing arithmetic is checked: `slab_plan`, `plan_bytes`, and `workspace_size` return `ok = false` on overflow or a negative capacity, and initializing with such a plan is `Invalid_Input` (`core.storage.plan`).

Every time a bank is cleared to be built again its revision rises, and a `Document` records the generation, the bank, and the bank's revision. Passes alternate banks inside one generation, so the revision is what keeps a handle to bank A invalid after a second pass has rebuilt bank A. Re-initializing a workspace advances its generation too. The staged procedures also check that the document belongs to the workspace they were given (`core.document.foreign`).

Small vocabulary, ordinary Odin

Use `Type_Name` for exported types and `verb_noun` for procedures. Call sites use a short package alias, as in `gd.convert`. Borrow the clarity of raylib's small verbs and explicit lifecycle, while avoiding hidden global engine state. There are no fluent builders, class hierarchies, required macros, or allocator defaults.

Inside this repository packages import each other by relative path, so a plain `odin build cmd/guidedog` works. Embedders may vendor `lib/` and map a collection.

```
import gd "lib/core"
import "lib/defaults"

// The host supplies typed backing slices for both snapshot banks, scratch
bytes,
// source slots, diagnostics, and an output byte buffer.
ws: gd.Workspace
if r := gd.init_workspace(&ws, storage); r.status != .Ok do return
defer gd.reset_workspace(&ws)

result := gd.convert(&ws, defaults.registry(), gd.Request{
  source   = {name = "intro.md", text = "# Welcome\n"},
  reader   = "commonmark",
  renderer = "html",
  policy   = gd.default_policy(),
  limits   = gd.default_limits(),
}, output[:])
if result.status == .Ok {
  html := output[:result.written]
  // Consume html before reusing the output array.
}
```

Initialization borrows storage; reset invalidates views and clears used-counts. No close or destroy procedure is needed because the library owns no resources. `convert` starts a new workspace generation and invalidates earlier workspace views, even when the new call fails. It returns output length, report span, and a by-value primary failure record. Callers needing inspection use the staged interface.

```
Status :: enum u8 { Ok, Invalid_Input, Unsupported, Policy, Capacity, Limit,
                  Cancelled, Internal }
Region :: enum u8 { None, Nodes, Text, Attributes, Lists, Links, Cells, Sources,
                  Scratch, Reports, Output, Source_Map }
Capacity_Need :: struct { region: Region, minimum: u64, exact: bool }
Result :: struct {
    status: Status,
    written: int,
    reports: []Diagnostic,
    primary: Diagnostic,
    need: Capacity_Need,
}

init_workspace      :: proc(ws: ^Workspace, storage: Storage) -> Result
reset_workspace     :: proc(ws: ^Workspace)
convert             :: proc(ws: ^Workspace, registry: Registry, request: Request,
                          output: []u8) -> Result
read_document       :: proc(ws: ^Workspace, reader: Reader, request: Request) -
> Read_Result
transform_document  :: proc(ws: ^Workspace, doc: Document, passes: []Pass,
                          limits := Limits{}) -> Read_Result
check_render        :: proc(ws: ^Workspace, doc: Document, renderer: Renderer,
                          request: Request) -> Result
render_document     :: proc(ws: ^Workspace, doc: Document, renderer: Renderer,
                          request: Request, output: []u8) -> Result
```

Returned `Document` values carry the workspace, its generation, the bank, and the bank's revision. `view` and the staged procedures reject stale handles and handles of another workspace. Render checking and emission use the same immutable snapshot, renderer, and options; emission repeats the check. Applications must not mutate exposed read-only-by-contract slices. Odin does not enforce this borrowing model.

`Read_Result` carries the same status, reports, primary failure, and capacity fields as `Result`, plus a document valid only on success. `Request` includes limits, policy, render options, optional passes, a resource provider, and a cancellation probe. Defaults come from explicit `default_limits` and `default_policy` procedures; zero-valued limits do not silently mean unlimited. `default_request(name, text, reader, renderer)` assembles a complete request from them.

A reader's own settings travel in `Read_Options.extension`, a `Read_Extension`: the caller's pointer and its typeid, made by `read_extension(&settings)`. A reader takes them back with `reader_settings(ctx, T)` (or `no_reader_settings(ctx)` when it takes none), which refuses a pointer of another type with `Invalid_Input` and a diagnostic naming both types' packages (`core.read.settings_type`, `core.read.settings_none`) before anything is read. Settings meant for one reader are therefore never reinterpreted as another's, which a bare `rawptr` allowed: `rst.Settings` passed to the Markdown reader returned `Ok` and read their memory as `commonmark.Settings`. The mechanism stays generic, so readers of other registries use it with their own types. `lib/defaults` adds typed helpers for the built-in readers (`rst_options`, `sphinx_options`, `myst_options`), which the compiler checks as well.

Resources are provided, not fetched

An RST `include` names a file only when the parser reaches it, so a host cannot supply every resource before parsing. `Request.resources` is a `Resource_Provider`: a procedure value and caller-owned user data. The reader asks for a name relative to the including source; the provider returns borrowed UTF-8 bytes or a refusal. The provider belongs to the caller's trust boundary and may allocate; the core never learns a path.

The core records each provided source in a caller-sized source table, so every span names its origin. Include depth, cycles, and total included bytes are bounded. A missing provider, a refused name, and a cycle have distinct diagnostics. The CLI provider confines reads to approved roots with symlink-aware checks, as GDS 0002 says.

Capacity is a result, not an exception

We do not promise to determine parse storage from input length alone. Nested constructs, decoded text, tables, and extensions can expand differently. The host provides capacities and an overall budget. A failing append reports the exhausted region and the minimum needed at that point. This is a lower bound unless `exact` is true. Arithmetic overflow is checked before addition, multiplication, or slicing.

A host may restart a pure conversion with more storage, subject to its budget. Retries start from input bytes; no partial parser state is resumable in version one. The CLI allows at most three retries and reports when the next size would exceed the chosen budget. That policy is outside the allocation-free library.

A renderer has one encoding procedure and three emitter modes: **check** reports unsupported content without output, **measure** counts bytes, and **write** stores them. The engine runs check and measure together, then write. An undersized output buffer yields its exact byte requirement before modifying output. Parsing or render checking failure also leaves output untouched. An unexpected failure or cancellation during emission returns `written = 0`; callers treat all output bytes as invalid. A successful write checks that actual bytes equal the measured count.

Callbacks and custom passes are part of the caller's trust boundary. Core cannot guarantee that an arbitrary caller-supplied procedure performs no allocation. There is no streaming sink API in the first release; partial writes and backpressure deserve a separate contract.

Conversion sequence

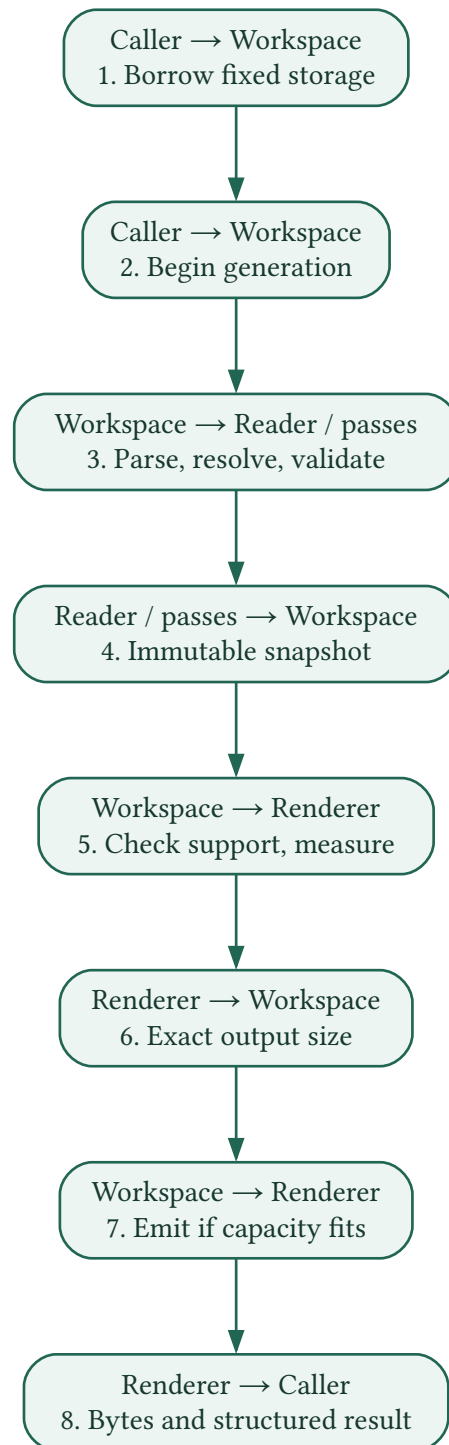


Fig. 7 No participant requests storage from an allocator during conversion.

Failure at steps 3 through 6 returns a diagnostic and prevents emission. After the caller consumes the result, reset or another conversion releases all borrowed workspace views together. Concurrent callers use separate workspaces.

Proving the allocation boundary

Tests install a failing allocator in both `context.allocator` and `context.temp_allocator`, then exercise successful and failing conversions; any allocator call fails the test. `tests/allocation_test.odin` converts one document with every built-in reader (RST, Sphinx, MyST, strict CommonMark, and Sphinx Markdown) into every renderer, across the options that choose other code (`Math_Style.Plain` and `.Sphinx`, permalinks, a highlighter), with and without a rebuilding pass, and checks that the output still holds every piece of content, including a heading, an admonition title, and a citation label longer than any fixed buffer. A refused allocation that code ignored would otherwise surface as quietly missing text, as it once did for Sphinx-style `display math`.

Counting context allocators alone is insufficient evidence, so `tests/boundary_test.odin` also audits the source of every package inside the boundary, on tokens so comments and strings never count. The boundary is derived, not listed: the test follows the repository imports of the packages whose code runs inside a conversion (`lib/defaults`, which assembles the readers and renderers, and `lib/highlight/guidedoc`, the highlighter renderers call) transitively, less declared host packages, so `core`, the readers, the renderers, and `lib/highlight` are covered, and a package a conversion comes to import is covered without editing the test. Each may import only an allowlist of Odin packages (`base:intrinsic`s, `base:runtime`, `core:unicode`, `core:unicode/utf8`, `core:slice`, `core:strings`, and `core:encoding/entity`), and from those only procedures that do not allocate, such as `strings.index` but not `strings.clone`; it may not call `make`, `new`, `append`, `delete`, `free`, or the other allocating built-ins, declare dynamic arrays or maps, or name the context's allocators. Test files are outside the boundary. Host and Typst costs are measured separately.

Allocation-free helpers that a host also wants in an allocating form keep the pure form in `core` and the allocation in the host: `normalize_nfd_into(dst, s)` writes Unicode NFD into caller storage (measuring with a nil `dst`), and the project host allocates for its index keys.

Use Odin's native `#soa` slices for the AST and typed side tables. Callers supply backing arrays; initialization borrows their SoA slices. Never use `#soa[dynamic]`.

Representation follows the work

`reStructuredText` and `CommonMark` readers share a semantic document representation. The `HTML` and `Typst` renderers traverse it directly. There is no writer-lowering layer, capability-cost graph, alternative search, or loss optimizer. Each renderer states exactly how each supported construct is emitted. A missing mapping is a reported implementation gap, never a silent fallback chosen by a planner.

Native `Typst` follows a separate path. `Typst` is both markup and a programming language. Extracting all its semantics into a small Odin tree would require an additional evaluator and still discard target-dependent behavior. `Guidedoc` instead passes native `Typst` to the `Typst` toolchain for `HTML` or `PDF`, through an explicit adapter outside its allocation-free core.

The conversion routes

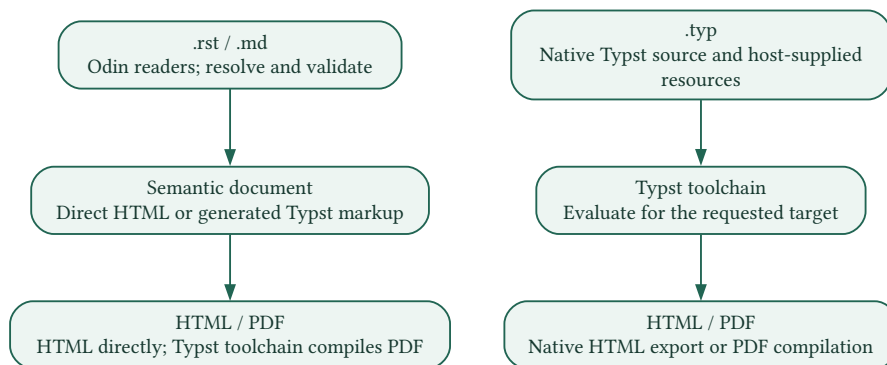


Fig. 8 Every PDF route uses Typst. Only RST and CommonMark require the Odin tree.

HTML exported from Typst must be checked against the pinned toolchain’s supported features. The product does not promise that arbitrary page geometry has an exact HTML equivalent. A target limitation produces an actionable diagnostic, with a direction to adjust target-specific Typst content. Native Typst HTML uses the compiler’s export behavior; it is not a home-grown conversion of Typst tokens.

Flat structure, meaningful types

Use one preorder node forest with a synthetic document root. `Node_Id` is a distinct `u32`; index zero is the root and the maximum value is reserved as invalid. `Text` is an offset and length into the snapshot text pool. `Source_Id` names an entry in the workspace source table, so spans in included files are distinguishable.

```

Node_Id :: distinct u32
Text :: struct { start, length: u32 }
Source_Span :: struct { source: Source_Id, start, end: u32 }
Node :: struct {
  kind:      Node_Kind,      // hot: scanned by traversal
  flags:     Node_Flags,
  subtree_end: u32,          // hot: exclusive preorder end
  value:     i32,            // small per-kind scalar: heading
  level, number
  text:      Text,           // leaf content, URL, label, or name
  payload:   u32,            // row in the kind's payload table, or NONE
  span:      Source_Span,
  attributes: Attribute_Range,
}
nodes: #soa[]Node // a tag scan reads nodes.kind without loading payloads
  
```

A subtree rooted at `i` occupies `[i, subtree_end[i))`. The first child, if present, is `i + 1`; the next sibling is the current child’s subtree end. Skipping a subtree is constant time. The builder keeps an open-node stack and patches a node’s end when it closes. Preorder does not mean that immediate children form a flat slice.

Index	Tag	End	Meaning
0	Document	5	Contains the entire document

Index	Tag	End	Meaning
1	Paragraph	5	Contains inline content
2	Text	3	“Read ”
3	Strong	5	Nested emphasis
4	Text	5	“carefully”

Rich payloads live in three typed SoA tables: **lists** (style, start, delimiter, tightness), **links** (destination, title, reference name, resolved node, link kind), and **cells** (row and column spans, alignment, header flag). Other per-kind data is either a scalar value or a keyed attribute row. Attribute keys are an enum — class, id, name, language, alt, width, height, scale, align, format, title, and a custom key carrying its own text — so common keys never copy strings. Resource bytes are supplied by the host and referred to by source ID; the core never fetches them.

Readers build skeletons in scratch and emit in preorder. CommonMark needs every link reference definition before inline parsing, and a paragraph may later become a setext heading or shrink when definitions are removed. Its reader therefore builds a block skeleton in scratch, then emits final nodes with inline content in one ordered walk. RST sections close when a later title of equal or higher level appears; the builder’s open-node stack expresses that directly. No reader patches a closed subtree. A Sphinx `only` directive may hold sections; the reader places it where its first title’s level belongs before opening it, so this holds there too.

Semantic coverage

The block kernel is: document, section, heading, paragraph, block quote, attribution, bullet and enumerated list, list item, definition list, definition item, term, classifier, definition, field list, field, field name, field body, option list, option item, option group, option, option argument, option description, line block, line, code block, doctest block, math block, raw block, thematic break, table, table head, table body, row, cell, figure, caption, legend, image, admonition, topic, sidebar, rubric, container, compound, footnote, citation, target, comment, substitution definition, contents, and a generic directive container for third-party extensions.

The inline kernel is: text, emphasis, strong, code, link, reference, image, footnote reference, citation reference, substitution reference, inline target, inline math, raw inline, soft break, hard break, subscript, superscript, title reference, abbreviation, and a generic role span carrying its role name.

The catalog grows with conformance fixtures when a specified construct needs a semantic distinction. The generic directive container is for third-party extensions, not an excuse to label standard RST unsupported. Cross-document references use stable project keys, not local node IDs.

No hash table may grow implicitly. Symbol tables use caller-owned scratch slots with a bounded load factor and capacity errors. Collision and probe limits prevent hostile input from causing unbounded work.

Resolution is a stage, not a rebuild

After parsing, the reader resolves names into side tables: named, anonymous, and indirect hyperlink targets; auto-numbered and symbol footnotes; citations;

and substitution definitions. A reference’s link row records its resolved node. A substitution reference points at its definition, and renderers emit the definition’s content in place. Nothing is copied or spliced, so resolution needs neither bank B nor a pass. Substitution cycles, duplicate targets, and unresolved references have dedicated reports with both source sites where two exist.

Reader conformance is explicit

reStructuredText means the language specification, implemented independently. The normative baseline is the Docutils reStructuredText markup specification with its standard directives and interpreted text roles. Guidedoc reproduces the language, not Docutils’ Python API, node classes, HTML writer, or message texts; docodin already provides that fidelity for projects that need it. Implementation phases do not redefine the baseline as a permanent subset. Release claims identify the tested specification revision and publish a coverage ledger for every construct, directive, and role, backed by Guidedoc’s own fixtures.

The ledger covers indentation, sections, transitions, all list families, literal and doctest blocks, line blocks, block quotes, grid and simple tables, explicit markup, comments, substitutions, targets, anonymous and indirect links, footnotes, citations, directives, roles, and escaping. Readers preserve source spans through substitution and reference resolution. Include cycles, substitution cycles, duplicate targets, unresolved references, and invalid nesting have dedicated reports.

Some directives require authority or a service: `include`, `raw`, `images`, `math`, and external resources. Recognizing and validating their syntax is part of full support. Executing them is governed by explicit policy. A disabled include is reported as a policy refusal, not mislabeled a parse error or silently discarded. No shell or Python runs because a document requests it. Sphinx-specific domains and custom directives are an extension layer and do not belong to the base RST conformance claim.

Markdown means CommonMark 0.31.2. Run its complete example suite. GFM tables, footnotes, task lists, and strikethrough are not silently included in the base mode. Optional extensions require explicit names, separate fixtures, and documented interaction rules. HTML blocks and inline raw HTML are parsed according to CommonMark even when publication policy refuses them.

Typst means native Typst markup and its language semantics. Do not invent a “Typst-lite” parser. Compiler diagnostics are adapted into the same report model. Native source maps to itself; generated Typst carries an interval map from generated ranges to original RST or CommonMark spans.

Direct rendering rules

Each renderer implements one documented mapping per semantic construct. A section becomes an HTML `section` with a heading, or a Typst heading; a footnote becomes an HTML `note` with backlinks, or a Typst footnote. A table with spanning cells preserves spans in both targets.

Every renderer routes node kinds with one exhaustive `switch` over `Node_Kind`: a new kind does not compile until each renderer says which of its parts writes it, or that it writes nothing (a comment, a substitution definition). A part given a kind it was not written for reports `core.render.kind (Internal)` instead of dropping the node and its subtree, and an integration test renders a document holding every

kind with every built-in renderer. Payload tables (`payload_of`), content models, and translation units are decided the same way.

Math. RST and CommonMark math is LaTeX notation; Typst math is a different language. HTML carries the LaTeX source in a `math` class for a client-side service. The Typst renderer uses a bounded LaTeX-to-Typst math adapter for fractions, roots, scripts, Greek letters, operators, delimiters, accents, matrices, and common symbols. An unknown command is an explicit `Unsupported` diagnostic naming the command; policy may instead emit the source verbatim with a warning. It is never silently dropped.

Raw HTML cannot be executed as Typst. The PDF path refuses target-specific raw content by default and points to a portable construct or a target-specific alternative. An explicit omission policy may warn and omit it; this is a fixed policy, not writer lowering. Standard semantic constructs remain required mappings.

HTML text, attribute values, and URLs have separate escaping rules. URL policy rejects executable schemes. Raw HTML is refused by default unless trusted-content mode is selected; the CLI exposes this as `--raw=refuse|omit|allow`. The CommonMark conformance harness uses a reference-rendering profile so safety policy is not confused with parser conformance.

Generated Typst uses a fixed trusted preamble and escaped data. Ordinary source text never becomes executable Typst code through interpolation: every markup-significant character is escaped, and code, URLs, labels, and resource paths have dedicated string-literal encoders. A user's native `.typ` file is executable Typst content within the configured toolchain environment.

Validation and cost

Validation checks column capacity, used bounds, root coverage, nested and strictly advancing subtree ends, legal parent-child categories, payload rows and text spans, source IDs, table geometry, and resolved references. Each finished document and rebuild must pass validation before rendering. Assertions are for internal bugs; untrusted data always receives a recoverable diagnostic.

For n nodes, t text bytes, and r references, normal traversal and emission are linear in $n + t + \text{output bytes}$. Reference indexing is expected linear within its probe budget. No implementation may promise linear time for the entire RST grammar without measurement. A work limit bounds repeated delimiter scans, reference processing, and substitution expansion.

A rebuild uses two banks and is linear in copied nodes and text. Version one favors a few clear stages over persistent trees, SIMD, or speculative parallel parsing.

Recursion is bounded by the depth limit and the stack budget

Decision. The builder's open nodes, validation, copying between banks, anchors, the HTML renderer's walk, and the CommonMark reader keep nesting in caller-provided stacks taken from scratch. The RST reader (a body inside a directive, list, or quote parses its own body), the Sphinx layer's C++ declaration parser (templates, declarators, and expressions nest), and the text and Typst renderers (a node renders its children) are recursive descent instead. Every one of those recursions passes through a guard (`enter`, `cpp_enter`) that checks two limits before the next frame:

- `Limits.max_depth`, the nesting the content may have, whose excess is `core.limit.depth`;
- `Limits.stack_bytes`, the machine stack the recursion may use, whose excess is `core.limit.stack`. A `Stack_Guard` (`lib/core/stack.odin`) records the address of a local where the reader or renderer starts, and each guard compares the address of a local in the current frame with it. The measure is exact whatever the procedures' frame sizes, the compiler, or the optimization level.

Both are `Limit` results; no input and no `max_depth` can overflow the stack. The default budget, `DEFAULT_STACK_BYTES`, is 512 KiB (0 means the default). Measured in the default build, the deepest constructs take about 1.3 KiB per level of nesting, so the default `max_depth` of 256 needs at most about 330 KiB and reaches its depth limit first; a build without optimization has frames about 2.7 times larger and may stop at the stack budget sooner, which is still a `Limit`. The thread running a conversion needs its caller's stack, plus `stack_bytes`, plus `STACK_RESERVE` (128 KiB) for the leaf work below the deepest guard (inline parsing, the math adapter's own bounded recursion). The default fits a 1 MiB thread, the smallest default thread stack of the supported platforms; Odin's threads on Unix take `RLIMIT_STACK`, usually 8 MiB.

The C++ parser also bounds its steps by the length of the declaration (64 steps per byte), since backtracking can retry alternatives at every level of nesting; a fold expression is tried only when its `...` is present, so nested parentheses parse in linear time.

Alternatives. Rewriting the recursive readers and renderers around explicit caller-provided stacks would remove the machine-stack dependence entirely, but touches most of their procedures for no change in behavior once the guard exists. Estimating a worst-case frame size per level and refusing a `max_depth` that cannot fit is fragile: the frames change with every edit, compiler version, and optimization level, and the estimate must hold for every path. Running conversions on threads sized from the limit needs thread stack sizes that Odin's `core:thread` does not expose; a host that wants more nesting raises `stack_bytes` and runs the conversion on a larger thread itself. This supersedes the earlier decision of this section, which bounded recursion by `max_depth` alone and made a host that raised it responsible for a larger stack.

`guidedog convert --stack-kib=N` sets `stack_bytes` for one conversion, beside `--max-depth`. `guidedog build` takes the same `--stack-kib`, `--max-depth`, and `--max-nodes`, and every session of the build (reading, translating, loading a cached document, linking and rendering a page) uses them; they are part of the digest of what changes reading and output, so changing one reads and writes again. The engine's limit reports name the `Limits` field to raise (`limits.stack_bytes`, `limits.max_depth`, `limits.max_nodes`), since the engine knows no command line; `lib/cli` replaces those hints with the flags that set them (`--stack-kib=N`, `--max-depth=N`, `--max-nodes=N`) when it shows the reports.

The largest budget is what the thread running the conversion holds besides the frames around the guarded recursion: `host.stack_budget_limit` leaves 2 MiB (and at least half the stack), so 6 MiB of Linux's and macOS's 8 MiB, and the 512 KiB default of Windows' 1 MiB. Convert runs on the main thread, whose stack is the soft `RLIMIT_STACK` on Unix (`host.main_stack_bytes`). A build reads on the

main thread, or with `-j` on threads `core:thread` starts, which it sizes to the soft `RLIMIT_STACK` too, or leaves at the system's default (512 KiB on macOS) when that limit is unlimited (`host.thread_stack_bytes`); pages render on the main thread. So a build's cap is the smaller of the two stacks' limits, and a larger `--stack-kib` is refused (`build.stack`) before anything is read. Without the option, the budget is the default, or less on a thread too small for it (`host.default_stack_budget`). Windows gives every thread the executable's 1 MiB.

The supporting libraries

The libraries Guidedog is built from may not import `lib/core` (the independence test lists what each may import), so each bounds its own recursion, with the same two ideas: a limit the input's nesting meets the same way on every platform, and, where the recursion's cost per level is not fixed, a measure of the machine stack itself. Each reports a clear error or leaves the construct out with a warning; none crashes.

- `lib/jinja`: `Options.max_nesting` (default 100) bounds the depth of the tree the parser builds, counting brackets, nested tags, and every link of operator, filter, test, attribute and `elif` chains, since chains build left-deep trees; every later walk over a template is bounded with it. `Options.stack_bytes` (default 512 KiB) is a frame-address guard started by the first public call on the environment and checked by the parser's nesting and every `eval` and statement, so macro calls, includes and recursive loops (`max_recursion`, 200) cannot overflow either. The defaults agree, so an ordinary template always meets `max_nesting` or `max_recursion` first and the guard is only the backstop for hostile ones: each procedure a level of nesting or recursion passes through keeps a small frame, dispatching through tables and leaving the rest of its work to helpers that are never inlined, so a level of recursion takes at most about 1.1 KiB in an optimized build and 1.9 KiB in a debug one (x86-64 Linux and arm64 macOS), and 200 levels fit in half and three quarters of the budget. The tests check both limits, and that margin, at every optimization level and under AddressSanitizer, whose builds get four times the budget. Walks over values (`repr`, `equality`, `ordering`, `tojson`) stop at 200 levels: `repr` prints `[...]` for a list that contains itself, as Python does.
- `lib/treesitter`: tree-sitter parses with a heap stack, so any nesting parses; `deeper_than` measures a tree with a cursor, without recursion.
- `lib/pyscan`: a module whose syntax tree nests deeper than `MAX_SYNTAX_DEPTH` (256) is not analysed and is marked `too_deep` (autodoc warns `autodoc.too_deep`); Python itself refuses 200 nested brackets or 100 indentation levels. Name resolution was bounded per path by `MAX_DEPTH` (48), but paths multiply (modules that each `import *` from two others, functions decorated several times with the next), so one lookup also has `MAX_STEPS` (100,000) steps and `LOOKUP_STACK_BYTES` (256 KiB) of stack, measured from its first frame, because an attribute of a class starts nested lookups through its MRO. MRO computations nest at most `MAX_MRO_DEPTH` (100) classes deep; a longer inheritance chain gets an incomplete MRO instead of cubic time.
- `lib/autodoc`: annotation evaluation (`MAX_EVAL_STEPS`) and value description (`MAX_DESCRIBE_STEPS`) count their steps, since aliases and constants that each use the next several times multiply the work; past the budget the annotation

or value is shown as written. One directive generates at most `MAX_GENERATED` (10,000) directives, with the warning `autodoc.limit`, since a class that contains itself under its own name is documented again at every level.

- `lib/odindoc`: Odin's parser (`core:odin/parser`) is recursive descent with no limit and takes 8 to 20 KiB of stack per bracket in a debug build, so an 8 MiB thread overflows near 600 nested parentheses, and `odindoc` cannot guard it from outside. Before parsing, it estimates the parser's stack from the tokens (brackets, pending type prefixes, and cast, ternary and `else` chains, each weighted by a measured frame cost) and leaves out a file past `PARSE_STACK_BYTES` (1 MiB), with the warning `odin.autodoc.nesting`. Odin's own core library estimates under 300 KiB. The `when` conditions `odindoc` evaluates are followed `MAX_WHEN_DEPTH` (64) levels deep.
- `lib/gettext` already bounded its plural expressions (100 levels, 128 nodes). Four costs a hostile catalog could square are now linear in it (or $n \log n$): a PO field's continuation lines are gathered in a builder and copied once, not concatenated line by line; an MO file whose strings share bytes is refused as damaged (sorting the table's spans finds any), so a small file cannot make every entry check and copy the same large string; the fuzzy search of `merge` walks a length-sorted, trigram-ranked candidate list that stops where no candidate could win, giving the answer comparing every pair gives, within a work bound of 256 bytes read per byte of msgids (`Merge_Stats` counts it); and wrapping a long line counts only the next piece. `lib/gettext/cost_test.odin` checks each by what it allocates or counts. `lib/highlight` keeps nested lexer states in an explicit stack of 24 frames; and `lib/graphdog` does not recurse (`Graphviz`'s parser reports DOT nested 10,000 deep as an error).

Alternatives. One shared guard package would remove the small repetition of the frame-address guard in `jinja` and `pyscan`, but would make every library depend on it; the guard is a dozen lines. Running Odin's parser on a thread sized from the file would avoid the estimate, but Odin's `core:thread` does not expose stack sizes.

Package structure and implementation order

Package boundaries express dependency direction; file boundaries keep procedures and explanations small enough to read in one sitting.

<code>cmd/guidedog/</code>	main.odin; composition and process exit only
<code>lib/core/</code>	package guidedoc; memory, AST, builder,
<code>validation, reports</code>	
<code>lib/readers/commonmark/</code>	CommonMark 0.31.2; myst-parser 0.16.1 syntax
<code>through a hook</code>	
<code>lib/readers/rst/</code>	RST blocks, inline markup, directives, roles,
<code>resolution</code>	
<code>lib/readers/sphinx/</code>	Sphinx directives, roles, domains, for RST
<code>and Markdown</code>	
<code>lib/render/html/</code>	semantic HTML encoding and escaping
<code>lib/render/typst/</code>	Typst source encoding, math adapter, source maps
<code>lib/render/text/</code>	plain text, as Sphinx's text builder writes it
<code>lib/defaults/</code>	explicit built-in reader and renderer registry
<code>lib/host/</code>	storage sizing, retries, resource provider,
<code>publication</code>	
<code>lib/typst/</code>	adapter to the Typst backend (installed CLI or
<code>the bridge)</code>	
<code>lib/cli/</code>	options, commands, terminal diagnostics
<code>lib/gds/</code>	metadata, lifecycle, registry, promotion, recovery
<code>lib/project/</code>	discovery, configuration, symbols, cache,
<code>builders, themes</code>	

lib/jinja/	Jinja templates for pages; independent of Guidedoc
lib/highlight/	code highlighting with Pygments' classes;
independent	
lib/highlight/guidedoc/	the highlighter as a <code>Render_Options.highlighter</code>
lib/graphdog/	Graphviz's C library for graphs; independent
lib/gettext/	gettext catalogs for translations; independent
cmd/graphdog/	a dot-compatible command over lib/graphdog
native/typst_bridge/	narrowly scoped Rust-to-C ABI bridge to the
Typst compiler	
native/graphviz/	script that builds Graphviz statically for
lib/graphdog	
tests/	conformance, malformed input, goldens, integration
tools/	source-limit checks and documentation commands
docs/gds/	standalone discussions, index, template,
and archive	

core imports no reader, renderer, OS, terminal, or Typst package. Readers and renderers import core and nothing that allocates. Defaults imports those adapters and assembles a registry. Host and CLI compose the pure library; Typst remains an explicit host dependency. Embedders can import core and a selection of adapters without the CLI. Guidedog, the application, is `cmd/guidedog` with `lib/cli`, `lib/project`, and `lib/host`; Guidedoc, the engine, is `lib/core` with the readers and renderers. The libraries a Guidedog build also uses (Jinja, the highlighter, graphdog for Graphviz, and gettext) import no Guidedog or Guidedoc package, so other programs can use them alone; only the small `lib/highlight/guidedoc` adapter joins one to the engine. `tests/independence_test.odin` checks every one of these import rules, and `tests/examples_test.odin` type-checks every program under `examples/` and every complete program in the cheat sheets (`README.md`, `lib/**/README.md`) and the guide (a Markdown block fenced as `odin`, or an `odin` code block), so the documented examples cannot fall out of date with the API. A block marked `fragment` is compiled too, completed by one convention: its own `import` lines stay at file scope and the rest becomes the body of `main`, so a `fragment` names its imports and declares the values the prose around it describes. Examples in doc comments, an indented block after a line “Example:” (Odin’s convention, which `odindoc` renders as a code block), are compiled the same way, and code indented in a comment without that marker fails the test. `tests/api_docs_test.odin` requires a doc comment on every procedure the libraries export, which states its contract as GDS 0001 asks: what it borrows or copies, how long results live, and how it fails. Memory has a checked shape, by each package’s convention: a package that allocates nothing (core, the readers, the renderers, defaults, highlight) may not call an allocator anywhere; in one whose procedures allocate, a procedure whose own code allocates, frees, or takes an allocator names who owns the result (the caller, an arena, the allocator, or what it borrows); and a package that runs entirely in the caller’s arena (`cli`, `gds`, `pyscan`, `autodoc`, `odindoc`) says so once, in its package comment or its README’s “Memory” paragraph. What only the core uses is private.

The readers and renderers are narrowed the same way. Each exports its entry points (`VERSION`, `CONFORMANCE`, the descriptor procedure, and the `read`, `read-inline`, or `render` procedure), its settings types with their defaults, and, where another package builds on it, a named interface: the RST reader’s extension interface (the directive and role handlers, the parsed directive and its options, the parser’s `node`, `scratch`, `line`, and `report` procedures, and the host interface through which MyST runs the same directives), the inline parser’s hooks, the CommonMark

reader's extension interface, and the Sphinx layer's configuration, MyST settings, and autodoc host interface. Everything else is private: a file that holds only implementation is `#+private`, and an implementation declaration beside public ones carries `@(private)`. Diagnostics are identified by their stable `code`; the `Message` values behind them are private, except the one a dialect reports itself (`rst.DIRECTIVE_UNKNOWN`). Package-internal tests still reach private declarations; a test in another package uses the public surface. `lib/README.md` ("Reader and renderer surfaces") lists each surface.

Every other library is narrowed the same way, so each can be used on its own through a small API that its README cheat sheet lists: `cli` exports only `run`; `gds` its record workflow (load, plan, lock, apply, recover, build), with the path and string helpers it once exported moved to their users; `highlight` its lexing, writing, style, and problem procedures, with the lexer engine private, and `highlight/guidedoc` only the two highlighters; `treesitter` and `graphdog` their procedures, with the C functions behind them private; `pyscan` the analysis `autodoc` builds on, with the scanner and analyser private; `autodoc` and `odindoc` their sessions or projects and one `document` procedure; `jinja`, `gettext`, and `i18n` the API their cheat sheets document, with tables, limits, messages, and the members behind procedure groups private; `inventory` and `fetch` their load, find, write, and get procedures. Where the narrowing found an unclear owner, the contract was fixed rather than described: `fetch.Result` is now always the caller's, with `fetch.destroy`, and `i18n.figure_for_language` always returns a string the caller owns. `typst` exports its backends, `locate`, the two compiles, and `offset_of`, with the bridge's C types private. `lib/host` and `lib/project` are narrowed by the same rule, below.

Host surface

`lib/host` is narrowed by the same rule. It exports what the command line, the project builder, the engine's test suites, and an embedder need to run the engine with memory and files: sessions that size, grow, and retry storage within a budget (`Session`, `run`, `read`, `session_destroy`, `output_text`, `describe_all`, `budget_report`), the steps of a session for a caller that loads saved documents (`plan_session`, `grow_session`, `prepare_session`, `resize_output`, `resize_source_map`, `read_kept`, `read_owned`), storage sizing without a session (`sizes_for`, `make_storage`, `destroy_storage`), budgets (`Budget`, `budget_join`, `budget_leave`, `Ledger`, `charge`, `discharge`, `settle`, `budget_problem`, `Meter`, `meter_allocator`, `metered_refusal`), files (`read_source`, `read_file`, `same_contents`, `each_chunk`, `io_problem`), includes and confinement (`Files`, `files_provider`, `canonical_roots`, `confined`, `confinement`, `containing_root`), publication (`publish`; `write_unsynced`, `sync_all`, `folders_of` for a commit of many files), diagnostics (`Report`, `Problem`, `describe`, `exit_code`, `display_path`, the text and JSON renderers, `excerpt_at`), interrupts, stack budgets, and the `guidedoc` convert route (`convert_artifact`). Budget reservations, region growth, staging files, terminal escaping, and byte counts as text are private.

One ownership rule covers the package: what a procedure returns is allocated in `context allocator` and belongs to the caller, which frees it with that allocator's arena; arguments are borrowed for the call. The exceptions say so in their contracts: a session owns what its results refer to until it runs again or is destroyed, and a resource provider or meter borrows its state for as long as it is

used. Failure is a `Problem` or `false`, never a panic; a budget refusal is `host.budget`, a system refusal within the budget `host.memory`. `lib/host/README.md` lists the surface; `tests/api_docs_test.odin` requires a contract comment on each procedure in it, and `tests/surface_test.odin` requires each exported name to appear in the cheat sheet, so the surface cannot grow without being advertised. The project builder’s surface is recorded with the project (the `0006-sphinx-replacement` draft, “Project surface”).

Sessions remember the allocator that supplied their current regions, so changing the calling context cannot change who frees them. Project steps and CLI conversions use reclaiming allocators for those regions: retries release actual storage rather than only returning a budget charge while an invocation arena retains the memory. Buffer replacement reserves the complete new buffer while the old one remains live, then frees the old buffer before returning its charge. The same budget and policy govern initial preparation and subsequent growth. `read_owned` inputs belong to the session, including their ownership records, and are freed before their reservations are released. `read_kept` preserves its original contract: the caller owns and frees the input bytes, and the session releases their charge when it is destroyed. Cached objects use `read_owned`, so discarding a stale cache frees its real storage immediately.

Diagnostic coordinates count Unicode scalars in the original source. Terminal excerpts are bounded windows around the error span, with separate cell widths for escaped controls, combining marks and wide characters. JSON retains the complete original source line, so clipping or escaping cannot change an editor’s coordinates.

`odin run tools/check` enforces the source limits: lines of at most 99 characters, never more than 108, files of at most 1,408 lines, and, for Odin, procedures of at most 70 logical lines. The line and file limits cover all implementation source, including the CSS, JavaScript, HTML templates, Typst templates (such as the quickstart book template), shell scripts, Rust, and TOML shipped with the code. Documentation is exempt: Markdown, reStructuredText, and Typst prose under a `docs` directory (these records included) are written for readers, and third-party data under `data` directories is kept as published.

```
odin build cmd/guidedog -out:build/guidedog
odin test lib/core -out:build/core-tests
odin test tests -out:build/guidedog-tests
```

The Odin toolchain is `dev-2026-09:a2fb372b7`; Typst is 0.15.1.

Adapter contracts

A registry is an immutable pair of slices: reader descriptors and renderer descriptors. Stable IDs identify adapters: `rst`, `sphinx`, `commonmark`, `myst`, `html`, `typst` (the generated-source renderer), and `text`. A reader ID names one profile, never a default that settings switch: `commonmark` is pure CommonMark 0.31.2 and takes no settings, and `myst` is the MyST Markdown that projects write (their `.md` documents are read with it and the Sphinx layer’s directives and roles). The CLI maps `md` to `commonmark` and exposes `pdf` as a host route through the Typst renderer and backend. Native `.typ` input is a backend route, not a falsely advertised zero-allocation reader.

A descriptor holds ID, version, conformance status, and a procedure value. Options are typed: `Read_Options` carries the reader's own settings as a typed `Read_Extension`, and `Render_Options` selects standalone or fragment output, the reference profile, and a document title. Registry validation checks unique IDs, non-nil procedures, and required metadata; `convert` runs it before reading, and the staged procedures refuse a reader, renderer, or pass without a procedure. Adapters retain no workspace pointers between calls.

Core's exported names form two surfaces, and everything else is private to the package: the conversion surface (workspaces, slabs, requests and their defaults, `convert`, the staged procedures, results, diagnostics, and the read-only view), and the adapter and pass surface (descriptors, contexts, the builder, the emitter, scratch, stacks, buffers, symbol tables, anchors, and shared text helpers), with three services for hosts beside them (documents saved as bytes for a build cache, Unicode NFD, and line and column positions). Generation and bank bookkeeping, slab carving, the pipeline's helpers, the kind and payload tables, the fold tables, the object format's constants, and the diagnostics only core reports are internal. Procedures on the two public surfaces state what they borrow, how long results live, and how they fail; `lib/README.md` lists every exported name by surface, and a test fails when core exports a name the list lacks. A `Buffer` whose content does not fit sets a sticky overflow flag and counts the bytes it was asked for, so a caller measures into a zero buffer and then takes exactly that much scratch (`scratch_buffer`) instead of truncating into a fixed array.

A pass is a procedure plus caller-owned user data. It receives an immutable source snapshot and a destination builder; it returns unchanged, rebuilt, or failure. The engine runs passes in order and validates each rebuilt snapshot. The builder exposes `begin_node`, `end_node`, `add_leaf`, `add_text`, `add_attribute`, and typed payload procedures. Every procedure returns a checked result. A failed builder is poisoned until reset, so ignoring one capacity failure cannot publish a truncated document. Unbalanced nodes fail finalization.

`Storage` holds two `Snapshot_Storage` values, scratch bytes, source slots, and a diagnostic slice. Each snapshot contains native SoA node, attribute, list, link, and cell slices and a text pool. All lengths are capacities, with separate used-counts. Tables may have zero capacity when a profile does not use them; encountering such a construct yields a precise capacity result. No descriptor constructor allocates, and no hidden "default workspace" exists.

The host library offers `convert_artifact` with explicit source, target, resource provider, workspace sizing, staging destination, and optional Typst backend. Its allocation boundary differs from `gd.convert`: core conversion is zero-allocation; resource acquisition and Typst compilation may allocate. A caller requesting PDF without a backend gets an actionable result.

Diagnostics without allocation

```
Diagnostic :: struct {
  code:    string, // stable, static: "rst.reference.unresolved"
  severity: Severity,
  status:   Status,
  span:     Source_Span,
  title:    string, // static
  message:  string, // static template: "No target is named {0}."
  hint:     string, // static direction
```

```
    args:    [2]Diagnostic_Arg, // integer, source span, or static text
  }
```

Messages are static templates with at most two typed arguments, so constructing a report never formats text. A source-span argument is quoted by the host renderer from the source table. When report storage fills, `primary` keeps the first failure, a counter records suppressed reports, and a truncation flag is set. Capacity failures remain reportable because they need no storage beyond the by-value `primary` record.

Correctness before tuning

Native SoA storage, bounded loops, and explicit lifetimes are foundational choices. They do not imply a speed claim. First make the complete pipeline correct, safe, and understandable. Prefer a clear copy to a fragile alias. Bound CPU work and memory at all stages; report exhaustion before overflowing counters or indexing storage.

After end-to-end correctness is established, profile representative manuals and adversarial inputs. Record peak storage, high-water counts by region, CPU time, and output size. Optimize only demonstrated bottlenecks. SIMD, parallel parsing, and cache-specific tuning are deferred. No throughput target is a release requirement.

Acceptance matrix

Concern	Evidence required
Memory boundary	No allocator calls on successful and failing core paths; import audit; exact-capacity and one-short fixtures for every region.
AST safety	Malformed IDs, ranges, nesting, table spans, and stale generations fail cleanly; randomized bounded trees preserve validator invariants.
CommonMark	Every example from the pinned 0.31.2 specification; renderer policy tested independently of the reference-output profile.
RST	Coverage ledger for the specification, standard directives, and roles, each entry backed by a Guidedoc fixture for HTML and Typst output.
HTML / Typst	Escaping, links, tables, notes, and source maps have fixtures; generated Typst compiles; HTML structure is checked.
Native Typst	HTML and PDF routes, diagnostics, and pinned resources pass integration tests.

Concern	Evidence required
Diagnostics	Stable text and JSON fixtures; every failure has a direction; report overflow and out-of-capacity reporting remain usable.
Host transactions	I/O faults, overwrite refusal, cleanup, and preservation of prior published artifacts.
Resource bounds	Deep nesting, include cycles, expansion, large lines, and delimiter stress terminate within configured memory and work budgets.

Unit tests target semantics and invariants. Golden fixtures show deliberate output changes. Expected failures are results, never crashes. A PDF text check cannot replace visual review, and a screenshot cannot prove reference resolution.

Security considerations

The engine processes untrusted markup. Its normative safeguards are checked arithmetic and slicing, bounded parsing and expansion, generation-checked handles, validated references, and target-specific escaping. Host authority is supplied only through the resource provider. No core procedure reads paths or executes code. Native Typst requires the separate host containment contract in GDS 0002.

Backwards compatibility

There is no earlier Guidedoc release. The proposal does not promise compatibility with zenfmt’s Zig ABI, manifests, or serialized AST, nor with docodin’s Docutils-compatible node classes or wire format. Changes to a published node schema require a separate GDS and an explicit compatibility decision.

Alternatives considered

A growing arena was rejected because it violates the allocation boundary. A pointer tree complicates fixed storage and ownership. Native SoA retains a clear semantic model while exposing compact columns. A single forest with typed side tables simplifies source coordinates and mixed block/inline nesting; separate forests remain a possible measured redesign. Writer lowering was removed at the user’s direction.

Reusing docodin’s RST engine was considered. It is complete and measured, but it reproduces Docutils’ document model and output byte for byte and allocates from growing arenas. Guidedoc needs its own semantic tree under a fixed-storage contract, so it implements the specification directly and treats docodin as research. Borrowing input spans for node text was rejected in favour of an owned pool; mixed ownership made bank rebuilds and validation harder than the linear copy it saved. A custom Typst evaluator would duplicate the toolchain and is not proposed.

Reference implementation and open questions

The packages listed above implement this proposal. `lib/README.md` is the embedder's cheat sheet and `lib/core/AST.md` the normative node mapping; `examples/` holds runnable programs, one of which converts with static arrays under a panicking allocator. Beyond the staged interface, three raylib-style entry points make the common case one buffer and one call: `workspace_size`, `init_workspace_from_bytes`, and `convert_text`.

Evidence on 29 September 2026, on macOS arm64 and Linux x86_64:

Concern	Evidence
Memory boundary	Zero allocator calls in core, every reader, and every renderer , on successful and failing paths and across rendering options, with all content kept; a token audit of their imports and calls.
Capacity	For RST and CommonMark into HTML and Typst, every region's smallest working capacity succeeds and one less reports that region; output is exact.
AST safety	Malformed ends, nesting, payloads, stale handles, and illegal parents fail validation; builders poison on the first failure.
CommonMark	652 of 652 examples of 0.31.2 in the reference profile; 38 hostile patterns end as results within the work limit.
RST	253 tests over blocks, directives, inline markup, roles, and resolution ; coverage ledgers in <code>lib/readers/rst</code> and its <code>inline</code> package; 1,500 fuzzed documents end without internal errors.
HTML / Typst	Escaping, policies, notes, tables, and anchors tested; generated Typst for every node kind compiles to PDF and HTML; 35 hostile strings round-trip.
Math	76 LaTeX translations compile; unknown commands are refused by name.

The RST ledgers now record multiline substitution names and generated Unicode punctuation support as implemented, with tests. The reader descriptor remains conservatively `Partial`; corpus agreement is not a claim of complete Docutils implementation or arbitrary extension compatibility. `List_Info.delimiter` is one byte, so the bullets `•`, `►` and `-` travel in a `bullet` attribute. Storage defaults are estimates refined by capacity retries, not measured profiles.

Review history

29 September 2026: initial Odin proposal. Review removed writer lowering, selected full RST and CommonMark, selected native Odin SoA, deferred optimization, and restored the standalone Guidedog Discussion format and inherited lifecycle.

29 September 2026, implementation review: state that RST is implemented from the specification rather than ported from docodin; make node text owned by the snapshot pool; define scratch as a marked bump region; replace up-front resource entries with a provider; resolve references in side tables so bank B is optional; require readers to emit in preorder from scratch skeletons; define one encoder with check, measure, and write modes; add the LaTeX-to-Typst math adapter; use relative imports in-repository; replace the options schema with typed options; and specify static diagnostic records.

29 September 2026, implementation: record the implemented packages and evidence, the one-slab entry points, the `target-name` attribute for embedded aliases, and project links and anchor prefixes for multi-document books.

29 September 2026, engine review: remove the last allocations inside the boundary (Sphinx display math, NFD normalization) and prove the boundary with an option matrix and a token audit; report content that does not fit a bounded buffer instead of cutting it; size and carve slabs from one checked plan; add bank revisions and workspace checks to document handles; validate registries in `convert`; make core internals private and name the two public surfaces; add `default_request` and typed read options; type-check the examples; extend the source limits to all implementation source, with documentation exempt; and record that RST and the renderers recurse under the depth limit.

30 September 2026, review: bound recursion by a machine-stack budget (`Limits.stack_bytes`) as well as the depth limit, in the RST reader, the C++ declaration parser, and the renderers, so no input or `max_depth` overflows the stack; bound the C++ parser's backtracking; and route node kinds in every renderer through exhaustive switches. Later the same day: bound the supporting libraries the same way (Jinja nesting and stack, Python and Odin source nesting, the multiplying work of name resolution and autodoc expansion), make their closed dispatches exhaustive, and add `convert --stack-kib`. Later again: narrow every library to the API its cheat sheet documents, list core's exports by surface and test the list, check the memory side of every public contract by package convention, and compile every Odin snippet in the documentation, fragments and doc-comment examples included.

30 September 2026, API review: narrow the readers and renderers as the core was narrowed. Their implementation is private (`#+private` files, or `@(private)` beside public declarations); each keeps its entry points, settings, and the interface other packages build on, documents every exported procedure's contract, and the doc-comment test covers them. The Sphinx layer uses its own ASCII text helpers instead of the RST reader's.

30 September 2026, host API review: narrow `lib/host` to the services its callers use ("Host surface"), state the package's ownership rule, give every exported procedure a contract, and list the surface in `lib/host/README.md`, which a test holds the exports to.

Sources and design provenance

The following sources distinguish language facts from decisions made by this proposal. Web sources were checked on 29 September 2026.

- Neighboring `../zenfmt/docs/zds/records/0002-zenfmt-architecture.typ` and `0013-layered-document-ir.typ`: AST, facets, reports, and host separation. Writer lowering from that design is explicitly excluded here.
- Neighboring `../zenfmt/core/src/host.zig`, `options.zig`, and `report.zig`: inspected implementation contracts, not code to port verbatim.
- Neighboring `../docodin/docs/dod/0004-data-oriented-storage.rst`, `0012-parser-research-0-2-0.rst`, and `0013-source-spans.rst`: handles, SoA storage, and span rules measured in Odin.
- [Odin language overview](#): native SoA, slices, distinct types, procedures, and implicit context.
- [RST specification](#) and the accompanying [standard directives](#) and [standard roles](#).
- [CommonMark 0.31.2](#): the selected Markdown baseline and conformance examples.
- [Typst language syntax](#) and [HTML export](#).
- [Typst compiler crate 0.15.1](#): Rust compiler stages and resource environment; the proposed C ABI is ours.

Lifecycle event

2026-09-29: prediscussion → discussion. Assigned a permanent number and opened for discussion.

Lifecycle event

2026-10-11: discussion → accepted. Maintainer-requested implementation-status reconciliation; current supported contract reviewed and deferred scope stated explicitly.

Lifecycle event

2026-10-11: accepted → committed. Current supported design implemented; beta review fixes and limitations are recorded. Native library and integration suites with leak checks; sanitizer and embedded-backend validation; `docs/manual/evidence/beta-review-20261011.md`. Deferred features are not claimed as implemented.

Number: 0004
Title: Memory budgets and what a build may read
State: committed
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-30
Updated: 2026-10-11
Discussion: No external discussion URL assigned
Labels: architecture, build, security

Memory budgets and what a build may read

Abstract

A Guidedog build holds one memory budget. Everything Guidedog allocates for its work is reserved from it before it is allocated, with sizes summed under overflow checks: every session’s storage, output, source map, and their growth; the files it reads; the drawings Graphviz hands back; the book’s text. Work whose memory is known only as it is allocated (saved state, page layouts, the search index, API listings) is metered: charged before each allocation, and an allocation whose charge is refused is not made, so the budget is never exceeded, not even by the step that fails; what a build holds does not grow with its pages. Values a template builds are sized before they are allocated, and Odin’s parser, which cannot survive a refusal, runs on a bound reserved ahead. When the budget is reached, a build at a terminal asks whether to raise it, keep its working memory on disk (memory-mapped temporary files, bounded by a disk budget), or stop; a build without one follows `--memory` and never waits. Reading threads wait for each other’s memory but never all at once, so the budget cannot deadlock. Memory that native code allocates for itself (Graphviz, tree-sitter, Typst) is named as outside the budget, with the operating system’s limits as its bound. A build reads only what the project shares with its documents; `--untrusted` narrows that to the source folder, reads no file a graph names, and runs no Typst the project wrote. Typst runs in a root that holds only the folders the build may read. This record supersedes the budget and confinement parts of GDS 0002’s “CLI resource policy”.

Motivation and scope

An external review found the budget bounding less than it claimed: storage sizes could overflow into negative values; page rendering, to-do lists, and source maps allocated outside any budget, or from a budget of their own; files were read without being counted. It also found the trust model narrower than documented: `--untrusted` still honoured the project’s `include_roots` (`["../.."]` included a file outside the project); a graph’s `image` read any file and wrote its absolute path into the drawing, even under `--untrusted`; and Typst could read anything below the nearest folder holding the source, build, and include folders, often a home folder.

In scope: the host’s budget (`lib/host`), the project build (`lib/project`), graphdog’s file access (`lib/graphdog`), and the Typst adapter’s contract (`lib/typst`). Out of scope: time limits, and a worker process for Typst (GDS 0002, first open decision).

Specification

Checked sizes

`host.sizes_for`, `host.grow`, and `host.total_bytes` sum with overflow checks, as `core.slab_plan` and `core.plan_bytes` do, and return `ok = false` for a size that cannot be represented, a negative count, or a capacity result’s minimum larger than an `int`. A session treats such a size as its budget reached (status `Limit`, report `host.budget`: “needs more working memory than can be counted”).

One budget, reserved before allocation

`host.Budget` is a limit and what is in use. A build has one (`Build.memory`, 1 GiB, or `--budget=MIB`, 1 to 1,048,576). Every session a build opens is `Session{shared = &b.memory, limit = b.memory.limit}`: reading, rendering a page, the to-do list's copies of other documents, the gettext builder's extraction, and the source location a book's Typst error is traced to. A session reserves before allocating:

- storage, output, and source map, in `prepare_session`;
- output growth (`resize_output`) and source-map growth (`resize_source_map`) when a renderer asks for more; a render grows only through these, and a refusal reports the budget, never a silent allocation;
- the cached object it loads (`read_kept`), released when the session ends.

The system may still refuse an allocation the budget allows. Every host allocation of a session's regions, output, and source map, and of a file it reads, checks the allocator's error: `make_storage` returns `ok = false` and frees what it made, `prepare_session`, `resize_output`, and `resize_source_map` give back their reservation and return false, and the session reports `host.memory` ("OUT OF MEMORY": the budget allows it, the system could not provide it) rather than working in regions left nil; `read_file` returns the same problem rather than an empty file (`refused_allocations_are_problems`).

`host.Ledger` charges what one unit of work holds besides its sessions, before the memory is allocated, and `settle` releases it when the unit frees it:

- a document's read: its source (read again by the reading thread, so a source is held only while it is parsed; the digest is of what was parsed), its includes (`Files.ledger`; a refusal is the new `Resource.Status.Limit`, reported as `core.include.limit`), and the Python and Odin sources its generating directives read;
- a page: the graphs it draws (graphdog's `Reserve`) and the images embedded in them;
- a static file copied into the output, freed once written;
- the book's text (each chapter before it is appended), its template and preamble, and the PDF the embedded Typst hands back (`typst.Options.reserve`). The build's arena keeps these to the end, so their charge is never returned; the budget then says what the process holds.

`guidedog convert` charges included files to its session's own budget (`--budget`).

Bookkeeping and writing, metered

Some work's memory is known only as it is allocated: parsing saved JSON, laying a page out with its template, building the search index. It runs on a `host.Meter`, an allocator that charges a ledger before each allocation it passes to the heap or the build's arena. A charge the budget refuses is refused before anything is allocated: the allocation fails with `.Out_Of_Memory`, the ledger records the refused bytes (`Ledger.refused`), and the step, which stops at the failure, ends with the budget's report (`host.metered_refusal`), keeping its own report of where the memory was asked for (a template's line, say). The budget is never exceeded. (An earlier version recorded the refusal and made the allocation anyway, for libraries assumed not to survive a failed allocation; the review showed a 4 MiB build allocating a 60

MB string before it failed. See “Refusal before allocation”.) What a build counts this way:

- Saved state (`load_json`): the environment, the previous output’s manifest, and a commit journal are read into the heap within the budget, freed once parsed, and what parsing allocates is metered and kept charged to the end. Saving them (`save_json`) encodes in a metered arena freed once written. A budget that cannot hold the saved state fails the build before it reads, rather than reading everything again.
- Publishing: what carrying out a commit needs beyond its journal (the paths a replaced record is put back through, room in the build’s reports for the commit’s warnings, buffers for the messages its steps write) is made before the journal is saved (`plan_commit`), and a refusal there decides nothing. Once decided, the commit allocates nothing until it is settled: its file operations make their paths in stack buffers and call the system directly (`host.rename_path`, `host.remove_path`, `host.sync_each`, `host.exchange`, `host.identity`), so a refusal can neither stop it halfway nor pass for a failure of the disk. Removing the replaced output afterwards is not a step of the commit.
- Files compared or digested: `emit` compares a file already in place with what would replace it through a 16 KiB stack buffer (`host.same_contents`), and a document’s included files, MathJax’s configuration, and the templates are digested the same way (`file_digest`), so none is held.
- Writing (`Scratch`): a page’s layout (its context, navigation, the template’s work, and the HTML it makes), the single page, the index pages and the inventory, the search index (whose words files are read with `host.read_file` and freed per document), a catalog template, and a compiled catalog are each made in a heap arena metered to the budget and freed once written. A page is written while its session still holds its rendered text, so that text is never copied; `singlehtml` and the book, which keep each page’s text, charge the copy (`render_document_page`’s `keep`) and their growth first. What outlives a step (its reports, its problem, the paths the generation records) is copied to the build’s allocator.
- API pages: listing the packages of `odin_autoapi_dirs` loads each package on its own in a metered arena, freed once its name, synopsis, and platforms are copied out.
- Inventories: everything `intersphinx` loads (fetched, cached, or local) is metered and kept charged.
- Catalogs: a PO or MO file is read within the budget and freed once parsed; the catalog, in an arena of its own, is charged its size once parsed, and each document’s messages the `gettext` builder extracts are charged as its template grows.

What a build holds therefore no longer grows with the number of pages it writes.

Refusal before allocation

Code behind a meter must handle a failed allocation, and every library it calls must survive one; each was checked by refusing, one test run at a time, every allocation it makes (the first, the second, ... and each from there on), and by a build that raises its budget from nothing to what it

needs in small steps, so that the refusal falls in every metered step in turn (`builds_stop_cleanly_wherever_the_budget_ends`):

- Arenas. `core:mem`'s `Dynamic_Arena`, after its block allocator refuses, keeps `bytes_left` at a block's size with a nil block, and hands out the next request at address 0. `host.Arena` replaces it behind meters: a refused block leaves the arena as it was. Its blocks are linked through a header and used through bounds-checked slices.
- Jinja (`lib/jinja`, ours) checks every allocation, stops a render with a `.Memory` error that names the template line (formatted without allocating), and works out the size of what it builds before building it: string and list repetition, `range`, `join`, `~`, padding filters, `format` widths. `{{ "x" * 60000000 }}` asks once for 60 MB, which a 4 MiB budget refuses, and the build's report shows the line. Every list and dict names the allocator that owns it. A render notes the containers it does not own that a template stores into (`items.append(x)`, `d.update(...)`, `ns.a = x`). Before its scratch arena is freed, it copies what they hold of that arena, or of the environment's, into their owner. Inside the render, sharing is Jinja's; a macro kept past its render fails when it is called (`lib/jinja: stored_values_outlive_the_render`, on a memory that poisons what is freed).
- JSON (`core:encoding/json`) returns an error on every refused allocation of the records Guidedog saves; it runs behind the meter as it is.
- Odin's parser (`core:odin/parser`) dereferences the nodes it allocates without checking them and crashes on a refusal. It is kept behind a bound reserved ahead: before a file is parsed, `odindoc.File_Reader.reserve` asks for `parse_bound(n)`, 256 bytes for each byte of source plus 256 KiB, charged at once (`host.meter_prepay`); allocations then draw on it before charging anew, and what a file did not use goes back before the next (`host.meter_refund`). The factor was measured (`parse_bounds_hold`): 20 to 30 for ordinary code, and up to about 170 for the densest input (long operator chains, runs of semicolons, one-character list items), counting each list growth as a new allocation. A file the budget cannot hold is left out with a warning and the budget's report.
- `zlib` (`core:compress/zlib`, for inventories) allocates three Huffman tables without checking them; they come from a buffer on the stack sized for them, and the output grows in the caller's allocator, whose refusal ends inflation with an error.
- Our own libraries behind meters (inventories, catalogs, the index and navigation pages, digests) check what they allocate; a catalog is compiled whole or not at all.

When the budget is reached

`host.Budget_Policy` says what a budget does with a reservation it cannot hold, and that no unit will make room for: with nothing set, it is refused, as before. `on_refusal` is called (one at a time, while the budget's other reservations wait, with an allocator of its own so it never charges the budget it decides about) and answers:

- raise the limit to a new value, and the reservation is tried again;
- spill: working memory past the limit (sessions' storage, output, and source map, and what meters allocate) comes from the policy's `host.Disk` from then on;

- stop, and later refusals do not ask again.

`spill` set from the start spills without asking. The command line sets the policy: `--memory=disk` spills, `--memory=ram` stops; without either, a build whose standard input and standard error are terminals asks, offering a budget that fits the free memory when what is needed does (twice the old one, or a quarter more than needed, in 64 MiB steps), and disk first otherwise. A build without a terminal never asks: it stops with the budget's report, whose hint names a concrete value such as `--budget=256` and, since the build has a Disk to spill to, `--memory=disk`. `guidedog serve` keeps an answer for its later rebuilds.

Before a build starts, the budget is compared with the memory the machine has free (`host.available_memory`: `MemAvailable` on Linux, free and inactive pages on macOS, `GlobalMemoryStatusEx` on Windows); a budget larger than that is asked about at a terminal (continue, lower it to nine tenths of the free memory, keep that much in memory and the rest on disk, or stop), and reported as a warning without one.

Working memory on disk

A `host.Disk` hands out memory from temporary files it maps, 64 MiB each, or a file of a larger request's own size. Before mapping a file it checks the disk budget (`--disk-budget=MIB`, `Disk.limit`) and the free space of the folder, which must keep 256 MiB (`DISK_MARGIN`) after it; then it writes the file out in full, so the space is taken before any of it is used and writing through the mapping never meets a full disk (which would end the process with `SIGBUS`). On Linux, macOS, and the BSDs the file's name is removed as soon as it is mapped, so the system reclaims it however the process ends; on Windows it is created to be deleted when closed and kept open until it is unmapped.

Use after unmapping is made impossible by ownership. A file counts the allocations it holds; it is unmapped when the last is freed, or, with every other file, by `disk_destroy`, which only the Disk's owner calls. The command line owns the Disk and destroys it after `project.release`, when nothing the build allocated, its reports included, is used any more. A header at the start of each file links it; it is read before the file is unmapped, and the file is unlinked from the list first.

What a build keeps for the whole build (files it reads into memory, catalogs, templates) is charged with plain ledger charges and stays in memory: those are refused past the budget even when working memory spills. Native memory (Graphviz, tree-sitter, Typst) is outside the budget as before, and outside the questions and the disk too.

Waiting without deadlock

Units that run at once join the budget (`budget_join`, `budget_leave`): the reading threads of `-j`. A reservation the budget cannot hold now waits while another joined unit is still running, since that one releases what it holds when it finishes. When every other joined unit is waiting too, nothing would be released: the reservation is refused, the unit's document fails with the budget's report, and its release lets the others go on. A unit that has not joined, or is alone, never waits. So no unit waits for itself, and waiting units never wait for each other; a budget too small for the work refuses some of it and says so.

Outside the budget, and its bound

Native code allocates for itself: Graphviz while it lays out (its input is bounded by the document; its output is counted), tree-sitter while it highlights, and Typst while it typesets, in the `typst` program or, with the embedded compiler, in Guidedog's process. Guidedog does not claim to bound these. A separate limit was considered: `RLIMIT_AS` counts reserved address space rather than memory used (runtimes reserve far more than they touch), macOS does not enforce it, and it cannot separate the embedded compiler from Guidedog. The bound is therefore the operating system's, for the whole build: a cgroup (`systemd-run --user --scope -p MemoryMax=...`), a job object on Windows, or a container. The compatibility guide says so. A worker process with its own limits remains GDS 0002's first open decision.

What a build may read

A trusted build reads below the source folder and the folders `conf.toml` opts into (`include_roots`, `autodoc_source_paths`, `odin_autoapi_dirs`), through `host.confined`, which follows links. Two readers that did not go through it now do:

- **Graphs.** `graphdog` gains `Options.files`: after parsing and before layout it shows the caller every file name a graph gives through `image`, `shapefile`, `imagepath`, `fontpath`, and `<IMG SRC>` in HTML-like labels (`label`, `xlabel`, `headlabel`, `taillabel`), and sets the path the caller returns, or refuses the graph (`.File_Refused`). The build resolves a name against the document's folder, accepts a PNG, JPEG, GIF, or SVG file inside the readable folders whose path needs no escaping, and leaves `imagepath` and `fontpath` out. After rendering, each path in the drawing is replaced by the image as a `data: URL`, so pages and books show it without a copy and no path reaches the output; a drawing that would still show a path is refused. A graph that may name files is drawn every time rather than taken from the graph cache.
- **Typst.** The book's paths stay relative to `book_root`, the nearest folder holding the source folder, the include roots, and the graphs folder (no longer the output folder, which Typst does not read). Typst runs in `typst_root`: a private temporary folder of links, at the same places, to only those folders, removed afterwards (the links, never what they lead to). A path in the book, its template, or its preamble then reaches those folders or nothing; the same holds for the embedded compiler, whose reads are relative to the root it is given. If the restricted root or any link cannot be made, the build stops with `typst.root` before compiling. It never falls back to the common ancestor. The hint explains temporary-directory permissions and Windows Developer Mode, or how to keep inputs inside one source root.

–untrusted

Before discovery, `confine_untrusted` removes from the build's configuration, each with a warning (`build.untrusted`), `include_roots` and every file or folder setting that leads outside the source folder: `html_static_path`, `html_extra_path`, `templates_path`, `locale_dirs`, `pdf_font_paths`, `autodoc_source_paths`, `odin_autoapi_dirs`, `html_logo`, `html_favicon`, `pdf_logo`, and `mathjax_config_path`. The build then reads only below the source folder. Ignoring

with a warning was chosen over refusing the build, so a project from elsewhere still builds, visibly without what it cannot have.

The configuration confined is the build's own copy (`build_config`), never the loaded project's: a program (or `guidedog serve`) that builds one project several times gets an untrusted build's confinement for that build only, and a trusted build afterwards reads the include roots again (`a_trusted_build_after_an_untrusted_one_is_not_confined`).

A graph under `--untrusted` reads no file: a graph naming one is shown as code with a warning.

The book runs only Typst that Guidedog generates: `pdf_preamble` and `_templates/book.typ` are left out with a warning, raw Typst is already omitted by the policy, and `pdf_packages` is offline. The generated Typst escapes everything a document says (text, math, strings), so a document cannot add Typst code; it names only images and graphs that passed confinement.

The Typst adapter's contract

`lib/typst` states what it guarantees instead of “not an untrusted-input sandbox”: Typst reads only paths below its root (a link below the root is followed), fonts from `font_paths` and the system's, packages from the network unless `offline` (embedded compiler only); its memory and time are its own. Typst code reaches everything below its root, so a caller compiling Typst it does not trust gives a root holding only what that code may read.

Rationale and alternatives

One budget per build, rather than one per session, is what bounds a build: a page no longer escapes it by opening a session of its own, and `--budget` means the whole build. Charging a ledger before allocating keeps the rule “reserve, then allocate” for memory that is not a session's. Re-reading a source in its thread keeps parallel reads from holding every pending source at once.

Refusing, rather than recording a refusal and allocating anyway, is what “reserve before allocating” means; it costs an audit of everything behind a meter, and a bound reserved ahead for the one library that cannot survive a refusal. An emergency pool that serves small allocations after a refusal was considered, and rejected: code that ignores one failed allocation would still dereference nil, and the pool would only move the limit.

Disk, not swap: the system swaps the whole process, Guidedog's code and every other program included, and only once memory is short; files mapped on request keep what is swapped to the build's working memory, bounded by a disk budget, and gone with the process. The files are written out in full first, rather than left sparse, so a full disk is a refusal before the fact, never SIGBUS when a page is first written. Asking at a terminal, and following flags otherwise, keeps scripts and CI from waiting on a prompt that nobody sees.

Waiting only while another unit runs replaces “wait only when holding nothing”, which reading could not keep once sources and includes are counted: a unit always holds its source while its session grows.

Embedding a graph's images, rather than copying them next to the drawing, makes one drawing serve pages and books alike, and keeps paths out of the

output. Links for the Typst root, rather than copies, cost nothing for large image folders.

Security and operational considerations

A build can be refused for memory it would have used before; the report names what needed how much and the flag to raise. Under `--untrusted`, native memory and time remain unbounded by Guidedog; the guide tells users to add the system's limits.

Backwards compatibility

Graphs keep drawing; those that named files outside the readable folders, used `imagepath`, or named non-image files now show as code with a warning. Books whose template or preamble read files outside the readable folders now fail with Typst's "file not found". `--untrusted` builds lose the settings listed above, with warnings. `Resource_Status` gains `Limit`. `graphdog.Options` gains `files` and `reserve`, `Error_Kind` gains `File_Refused`, and `typst.Options` gains `reserve`; zero values keep the old behaviour. `Build_Options` gains `memory` (a `host.Budget_Policy`) and `Build_Result` gains `peak` and `budget`; `Budget` gains `policy`; `odindoc.File_Reader` gains `reserve`; `jinja.Error_Kind` gains `Memory`. The command line gains `--memory=ram|disk` and `--disk-budget=MIB`. A metered step that used to finish over its budget now stops at the refused allocation, with the same report.

Acceptance criteria and implementation status

Implemented, with tests: checked sizes and budget growth (`lib/host: sizes_never_overflow`, `growth_is_reserved_from_the_budget`, `waiting_units_never_deadlock`, `file_reads_are_charged`); `graphdog's` file access (`the_caller_decides_every_file_a_graph_names`, `the_caller_can_refuse_the_output`); builds (`tests/budget_test.odin`: one budget bounds reading and rendering; files, includes, and a conversion's includes are read within it; `tests/trust_test.odin`: graphs, `--untrusted` roots, and books). The same integration suite runs against the embedded compiler and the external compiler. `book_root_failure_never_widens_readable_access` also forces a root preparation failure after a successful link and checks that no wider root is returned.

Implemented, with tests, too: bookkeeping and writing (`lib/host: meters_charge_before_allocating`; `lib/project/budget_test.odin`: saved state, search words, page layouts, listed Odin packages, and catalogs larger than the budget are refused with its report; files are compared and digested in chunks).

Implemented, with tests, too: refusal before allocation and the policy (`lib/host: meters_charge_before_allocating` counts what reaches the backing allocator, `arenas_survive_refused_blocks`, `meters_prepay_a_bound`, `policies_raise_or_stop`, `spilled_memory_goes_to_disk`, `disk_memory_round_trips`, `disk_memory_is_bounded`, under `-sanitize:address` as well; `lib/jinja: refused_requests_never_crash_a_render`, `large_values_are_refused_before_they_are_built`; `lib/odindoc: parse_bounds_hold`, `refused_bounds_leave_files_out`; `lib/inventory: refused_memory_never_crashes_a_load`; `lib/gettext: refused_memory_never_crashes_writing`;

`lib/project:` `builds_stop_cleanly_wherever_the_budget_ends` and `large_template_values_are_refused_before_allocation`, the review’s case, whose budget never holds more than 4 MiB; `lib/cli:` the questions with a scripted terminal, and the flags; `tests/memory_test.odin:` the command line without a terminal). The review’s reproduction in the September review peaked at 13 MB of resident memory, against 241 MiB before. The October candidate measurement is recorded in the beta review.

The October review adds `compiled_book_output_refusal_is_a_budget_problem:` refusing the copy of an embedded compiler’s PDF returns the memory problem, exit 3, with numeric budget advice. It does not misreport successful compilation as a launch failure. Runtime hints distinguish host budgets, renderer retry limits, and native compiler memory; `tests/hints_test.odin` rejects placeholder budget corrections.

Not counted: memory native code allocates for itself (Graphviz while it lays out, tree-sitter while it highlights, Typst while it typesets), bounded by the operating system as described above. Outside any build’s budget, because no build exists yet: `conf.toml`, read when a project loads, at most 1 MiB.

Open questions

- Should a build refuse, rather than ignore, `include_roots` under `--untrusted`, for projects whose documents cannot be read without them?
- Should the graph cache key drawings that embed files by the files’ digests, rather than draw them every time?

Review history

2026-09-30: drafted with the implementation, after the external review. Same day: the bookkeeping, API listing, catalog, and page-writing memory it listed as not counted is now counted (metered steps); only native memory remains outside. Same day, after the second review: metered allocations are refused before they are made rather than recorded and made; values a template builds are sized first; the Odin parser runs on a bound reserved ahead; and a build at its budget asks, or follows `--memory`, to raise it or keep its working memory on disk.

References

- GDS 0002, Guidedog: CLI resource policy (superseded in part by this record).
- GDS 0003, the Guidedoc engine: capacities and budgets.
- Graphviz 16.1, `lib/common/utls.c` (`safe_file`) and `lib/gvc/gvusershape.c`.
- Typst 0.15.1, the CLI’s `--root` and `typst-kit`’s file resolution.

Lifecycle event

2026-10-11: prediscussion → discussion. Assigned a permanent number and opened for discussion.

Lifecycle event

2026-10-11: discussion → accepted. Maintainer-requested implementation-status reconciliation; current supported contract reviewed and deferred scope stated explicitly.

Lifecycle event

2026-10-11: accepted → committed. Current supported design implemented; beta review fixes and limitations are recorded. Native library and integration suites with leak checks; sanitizer and embedded-backend validation; docs/manual/evidence/beta-review-20261011.md. Deferred features are not claimed as implemented.

Number: 0005
Title: An Odin domain and Odin API documentation
State: committed
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-30
Updated: 2026-10-11
Discussion: No external discussion URL assigned
Labels: architecture, build

An Odin domain and Odin API documentation

Abstract

Guidedog documents Odin APIs as Sphinx documents Python ones. An `odin` domain, always on, describes packages, procedures, procedure groups, structs, unions, enums, bit sets, bit fields, other types, constants, variables, foreign imports, fields, and enumerators, with signatures written and shown as Odin writes declarations; roles link to them; each object has an id, an index entry, a table of contents entry, and an `objects.inv` entry. Generating directives (`odin:autopackage` and five others) and API pages (`odin_autoapi_dirs`, as `sphinx-autoapi's autoapi_dirs`) write that domain's reStructuredText from Odin source, parsed with Odin's own `core:odin/parser`. The compiler never runs. Guidedog's own reference, `docs/api`, is built this way.

Motivation and scope

Guidedog is written in Odin, and Odin has no documentation tool that produces a cross-referenced site and book with prose around the reference. Sphinx has no Odin domain. The Python pieces Guidedog already has (the Python domain, `sphinx.ext.autodoc` from static analysis, the project's object tables and inventories) give the design; this record applies it to Odin.

In scope: the domain in the Sphinx reader (inside the allocation boundary of GDS 0003); `lib/odindoc`, a host library that analyses Odin packages and writes the domain's text; the project's generating directives, API pages, cross-references, inventory, and incremental rebuilds. Out of scope: type checking, evaluating constants, and running anything.

Specification

The domain

`odin:package` sets the current package, as `py:module` sets the module, with `:synopsis:`, `:platform:`, `:deprecated:`, `:no-index:`, and `:imports:`, the import

aliases the package’s signatures use (`gd=core rst=readers/rst`). A package is named by its import path: `core:fmt` for a collection, or its folder below a project root (`readers/sphinx`). `odin:currentpackage` changes the context without a target.

Directive	Signature, as Odin writes it
<code>odin:procedure</code> , <code>odin:proc</code>	<code>[@(attrs)] name :: [#force_inline]proc["c"](params) [-> results][tags][where ...]</code> or the short form <code>name(params) -> results</code>
<code>odin:procgroupp</code>	<code>name :: proc{a, b}</code> ; each member links to its procedure
<code>odin:struct</code> , <code>odin:union</code>	<code>Name :: struct(\$T: typeid) #packed,</code> <code>Name :: union #no_nil {A, B}</code>
<code>odin:enum</code> , <code>odin:bitset</code> , <code>odin:bitfield</code>	<code>Name :: enum u8</code> , <code>Name :: bit_set[E; u8]</code> , <code>Name :: bit_field u32</code>
<code>odin:type</code>	distinct types and aliases: <code>Handle :: distinct uintptr</code>
<code>odin:field</code> , <code>odin:enumerator</code>	inside a type: <code>name: T</code> with a tag or a bit size; <code>Name = 3</code>
<code>odin:const</code> , <code>odin:var</code>	<code>NAME :: 64,</code> <code>NAME : int : 64; name: T = value</code>
<code>odin:foreign</code>	a foreign import: <code>libc "system:c"</code>

Every object takes `:no-index:`, `:no-index-entry:`, `:no-contents-entry:`, `:no-typesetting:`, and the domain’s own `:package:`, `:private:`, `:deprecated:`, `:availability:` (a first paragraph “Availability: Windows, Linux.”), and `:foreign:`. Members nested in a type are named `Type.member`. Later signature lines of one directive are the same object on other targets: they get no id and no entry.

Signatures become `Desc_*` nodes: attributes and keywords as `Desc_Annotation`, the package’s short name as `Desc_Addname` (with `add_module_names`), the name, parameters as a `Desc_Parameterlist`, named results as a second one after the text “ -> “. Names in type positions become `Xref{odin:type}` with the context attributes `odin:package` and `odin:type`; an alias of `:imports:` is followed to its package in the reader. Built-in types, keywords, polymorphic names (`$T` and `T` after it, including a nesting type’s), array lengths, enumeration names, defaults, constants, tags, and where clauses stay text, so nothing that cannot be an object produces a broken link, the way the Python domain leaves built-ins to its builtin resolver.

Ids, references, and the inventory

An object’s id is Sphinx’s `make_id("odin-" + package + "." + name)`: `odin-core-fmt.println`, `odin-readers-sphinx.Config.docname`; a package’s is `odin-package-core-fmt`. `make_id` keeps case, dots, and underscores, as for Python ids, and turns the import path’s `:` and `/` into hyphens: ids are readable in URLs, the same in every build, and apart for packages that share a last segment. The alternative, `<package>.<name>` as for Python, would put `:` and `/` into ids.

Roles `pkg`, `proc`, `type`, `const`, `var`, `field`, `enumerator`, and `obj` take the Python domain’s `~` and leading `..` A target is looked up as `package.Type.name`, `package.name`, the name as written, then by the end of an object’s name after a `:`

or /, so `fmt.println` finds `core:fmt.println` as Odin code names a package by its last segment; a leading `.` searches any suffix, filtered by the role's object types. Index entries read “println (procedure in core:fmt)”. `objects.inv` lists every object as `odin:<objtype>`, packages with priority 0.

Generating from source

`lib/odindoc` (a host library; it imports only Odin's `core:` and `base:` packages) reads every `.odin` file of a package folder except `*_test.odin` and `#+ignore` files through the host's confined reader, parses each with `core:odin/parser`, and records declarations in source order: kind, signature as written (the source span normalized to one line, never a procedure's body), attributes, doc comment (the comment group directly above, and a field's or enumerator's comment at the end of its line), privacy (`@(private)`, `@(private="file")`, `#+private`), members, and targets.

`odin:autopackage`, `odin:autoproc`, `odin:autoproccgroup`, `odin:autotype`, `odin:autoconst`, and `odin:autovar` hand the request to the host, as `autodoc`'s directives do (`Config.odin_autodoc`), and parse the answer where they stand. Options: `:members:`, `:undoc-members:`, `:private-members:`, `:exclude-members:`, `:member-order:` (source, alphabetical, groupwise), `:no-index:`.

Doc comments are plain text and become `reStructuredText` that reads as written: paragraphs and lists stay; an indented block is `code-block:: odin` (text after “Output:”); “Inputs:” and “Returns:” items become `:param name:` and `:result name:` fields (a new `result` field of the domain) or `:returns:`; “NOTE:” and “WARNING:” lines are admonitions; backquoted code is literal; every other markup character is escaped.

Platforms

Files are read as Odin builds them for every target at once. File-name suffixes (`_windows`, `_linux_arm64`, `_amd64`) and `#+build` tags (negations, the older `//+build`) give a file's targets, a set of (OS, architecture) pairs Odin supports; file-scope `when` blocks comparing `ODIN_OS` and `ODIN_ARCH` narrow them. Declarations of one name and signature merge, with the union of their targets. One made for every target of the package is documented once; one made for some says where (“Availability: Windows”); one whose signature differs per target is documented once per signature, the first indexed. A package built only for some targets says so with `:platform:`. No setting is needed.

API pages

`odin_autoapi_dirs` lists folders, relative to the source folder; each package below them gets a page `<odin_autoapi_root>/<path>` holding `odin:autopackage` with `odin_autoapi_options` and `odin_autoapi_member_order`, and `<root>/index` lists the packages with their synopses in a hidden toctree. The pages are virtual documents: `discover` adds them to the project's sources with their text, the build reads them as it reads files, and they never appear in the source folder. They take part in every builder, search, the general index, and the book; `odin_autoapi_add_tocentry` adds the index to the root document's first toctree. `odin_collections` names folders for import-style arguments (`core:strings`) and gets no pages. All settings are checked when `conf.toml` loads, with Elm-style diagnostics.

Setting	Default	Meaning
<code>odin_autoapi_dirs</code>	<code>[]</code>	folders whose packages get pages
<code>odin_collections</code>	<code>{}</code>	collection names to folders
<code>odin_autoapi_root</code>	<code>"api"</code>	the pages' folder
<code>odin_autoapi_options</code>	<code>["members", "undoc-members"]</code>	options of each page
<code>odin_autoapi_member_order</code>	<code>"source"</code>	source, alphabetical, groupwise
<code>odin_autoapi_add_toctree_entries</code>	<code>true</code>	the index joins the root toctree
<code>odin_autoapi_generate_api_docs</code>	<code>true</code>	<code>false</code> : directives only

Every Odin file read is a dependency of the page that read it (`Doc_Info.deps`), so editing a package reads only its page again; the index's text holds the synopses, so its digest changes when one does. A page whose package was not found is read again on every build.

Rationale and alternatives

The Python domain's design was kept wherever Odin allows, so a Sphinx user needs to learn nothing new: the same options, the same info fields, the same `~` and `.`, the same nesting of members, and the same inventory. Signatures are Odin's own syntax rather than a C-like one, since Odin readers read declarations that way.

Parsing with `core:odin/parser` keeps the reader of the source the one the language ships, so every syntax Odin accepts is read. Running `odin doc` would give types checked by the compiler, but would run a compiler at build time, for one target at a time, and could not see the other targets' files; static analysis documents the whole cross-platform API.

Virtual documents, rather than files written into the source folder as `sphinx-autoapi` writes them, keep the source folder the author's and let incremental builds work on dependencies as `autodoc`'s pages do.

Security and operational considerations

Building never compiles or runs Odin code. Source is read only below `odin_autoapi_dirs` and `odin_collections`, through `host.confined`, as includes are, and within the document size limit. A package that does not parse is reported with a warning, and what could be read is documented.

Backwards compatibility

The domain adds directives and roles under `odin:`, a name Sphinx does not use; projects that do not use it are unaffected, and their documents read as before. The environment's version changed because documents may now be virtual.

Acceptance criteria and implementation status

Implemented: the domain (`lib/readers/sphinx/odin.odin`, tests in `odin_test.odin`, allocation-free reading in `reading_makes_no_allocator_calls`), `lib/odindoc` with

its tests, the project's generating directives, API pages, resolution, inventory, and dependencies (`tests/odin_test.odin`), and `docs/api`, which builds Guidedog's own reference with strict HTML and PDF builds. API page counts are build observations, not a fixed part of the domain contract. The beta review reruns the package and integration suites with allocation leak checks and AddressSanitizer.

Open questions

- Private types that public declarations mention cannot be linked; `docs/api` lists them in `nitpick_ignore`. Should the generator show them without a link, or should `lib/` make such declarations private?
- Whether `A :: B` declares a type or a constant is told from the names; the compiler would know.
- `when` conditions other than `ODIN_OS` and `ODIN_ARCH` comparisons are not evaluated.
- A package module index (`odin-modindex`), as the Python module index, is not written.

Review history

2026-09-30: drafted with the implementation.

References

- GDS 0003, the Guidedoc engine and its allocation boundary.
- GDS 0006, Sphinx-style projects and recoverable publication: domains, autodoc, and the native project model.
- Sphinx 9.1, `sphinx.domains.python` and `sphinx.ext.autodoc`; `sphinx-autoapi`.
- Odin's `core:odin/parser`, `core:odin/ast`, and its file-name and `#+build` rules.

Lifecycle event

2026-10-11: prediscussion → discussion. Assigned a permanent number and opened for discussion.

Lifecycle event

2026-10-11: discussion → accepted. Maintainer-requested implementation-status reconciliation; current supported contract reviewed and deferred scope stated explicitly.

Lifecycle event

2026-10-11: accepted → committed. Current supported design implemented; beta review fixes and limitations are recorded. Native library and integration suites with leak checks; sanitizer and embedded-backend validation; `docs/manual/evidence/beta-review-20261011.md`. Deferred features are not claimed as implemented.

Number: 0006
Title: Sphinx-style projects and recoverable publication
State: committed
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-29

Updated: 2026-10-11
Discussion: No external discussion URL assigned
Labels: architecture, cli, build

Sphinx-style projects and recoverable publication

Abstract

Guidedog is a documentation system for reStructuredText and Markdown that makes websites and PDF books. It follows Sphinx’s authoring workflow within a documented support boundary. It aims for clear operation, predictable output, and useful diagnostics. Sphinx is its inspiration, not its specification. Guidedog keeps what Sphinx’s users rely on (the project layout `guidedog quickstart` creates, `guidedog build html` and `guidedog build pdf` in the project folder, `toctree`, cross-references, domains, the index, glossaries, numbered figures, conditional content, and stable link anchors) and designs the rest idiomatically in Odin: declarative `conf.toml` in place of executed `conf.py`, declared object types in place of Python extensions, static analysis of source code in place of importing it, typed data and explicit state in place of runtime hooks. Where Sphinx’s way is worse, Guidedog differs and says so in the support matrix. Typst is the PDF layer: the book template, a preamble, and raw Typst blocks. Plugins come later and will use the extension points this record introduces. There is no Makefile: the command line works on the project folder directly.

Motivation and scope

GDS 0002 called Guidedog “Sphinx-like”, but its first project builder was a converter with a light project mode: a `guidedog.toml` unlike `conf.py`, navigation derived from relative links instead of `toctree`, no cross-document labels, no domains, no index or search pages, and no Sphinx builders. RST support was accepted as `partial`. That missed the intent. Guidedog should offer a clear migration path for supported authoring features, and a new project should feel familiar to anyone who knows Sphinx. Its own native workflow is the design target. It need not be a drop-in replacement: copying behaviour that exists only because Sphinx is a Python program that runs its configuration and extensions would import Sphinx’s weaknesses along with its strengths.

Principles

- **Capability, not imitation.** Supported authoring features follow Odin’s idioms. Arbitrary Python configuration and extensions remain outside the support boundary.
- **Data over code.** Settings and object declarations are checked data. Python configuration is never executed. Trusted PDF templates run in Typst; `--untrusted` substitutes Guidedog’s template and omits raw Typst.
- **Stable where users depend on it.** Page paths, anchors, `objects.inv`, and numbering match Sphinx, so links into and out of a project keep working.

- **Better where Sphinx is weak.** A failed build publishes nothing; diagnostics explain and are the same on every build; files outside the project are read only when allowed; memory is bounded.
- **Migrate, don't emulate.** Guidedog is a new system that supports Sphinx projects, with caveats. Sphinx fidelity is pursued so existing projects build, but it is not the final goal. Where Odin's idioms conflict with Sphinx's Python runtime (executed `conf.py`, extensions' `setup()`, event hooks, imported code), Guidedog does not emulate the runtime: it has a migration path instead. `guidedog migrate` converts literal settings and declared object types today; later, a Guidedoc migration plugin will help transfer supported Sphinx authoring features, and a project that cannot be migrated is told exactly what needs converting by hand.
- **Honest.** Every difference from Sphinx is documented, with its reason, in the manual's support matrix; a comparison harness against pinned Sphinx measures the rest.

This record supersedes the project builder of GDS 0002 (`init`, `build`, `serve` over `guidedog.toml`). GDS 0003 retains a conservative `partial` RST conformance descriptor; benchmark matches do not promise every possible input. The allocation-free Guidedoc engine, its readers and renderers, the Typst backends, and `guidedog convert` remain.

In scope: the command line, the project layout, `conf.toml`, builders, the build environment and incremental builds, the supported native directives and roles, the Python, C, C++, JavaScript, reStructuredText, and standard domains, generated index, search, and module-index pages, Odin's Jinja implementation, CommonMark and supported MyST syntax, Graphviz graphs, static Python autodoc (source parsed, never run), themes, and the PDF book. Complete Sphinx compatibility is not a goal. Out of scope: executing `conf.py`, other Python extensions (autosummary, napoleon, viewcode, and projects' own), and third-party themes that need Python code.

Specification

Compatibility target

The comparison point is Sphinx 9.1 with Docutils 0.22. For the same sources, Guidedog aims to produce: the same set of pages at the same paths (`docname.html`, or `docname/index.html` for `dirhtml`); the same document hierarchy from `toctree`; the same anchor ids for sections, labels, terms, and domain objects, so existing links keep working; the same numbering of sections, figures, tables, code blocks, and equations; and diagnostics for the same problems, explained in Guidedog's way. HTML markup and appearance come from Guidedog's theme, not Sphinx's; they are not byte-compatible and need not be. Where matching Sphinx would mean copying a runtime quirk or a weaker design, Guidedog does the better thing and the support matrix records the difference and why.

Command line

```
guidedog quickstart [DIR] [-q] [-p PROJECT] [-a AUTHOR] [-v VERSION] [-r RELEASE]
                    [-l LANGUAGE] [--sep] [--suffix .rst] [--no-batch]
```

```

guidedog build [BUILDER] [PROJECT] [options]      # make-mode: _build/BUILDER
guidedog build -b BUILDER SOURCEDIR OUTPUTDIR [FILES] [options] # sphinx-
build form
guidedog clean [PROJECT]
guidedog serve [PROJECT] [--port 8000]           # autobuild: rebuild on change
guidedog migrate [conf.py]                       # literal settings
to conf.toml

```

PROJECT defaults to the current directory. Guidedog finds `conf.toml` there, in its `source/` folder (the `--sep` layout), or in the nearest parent that has one. The source directory is the folder holding `conf.toml`; the build directory is `_build` beside it, or `build/` in the separated layout. `guidedog build html` is `make html`; `guidedog build` with no builder builds `html`. Options follow `sphinx-build`:

Option	Meaning
-a	write every output file, not only out-dated ones
-E	ignore the saved environment and read every document (fresh build)
-d DIR	environment cache directory (default: <code>BUILDDIR/.doctrees</code>)
-j N, -j auto	read and write documents in parallel
-c DIR	where <code>conf.toml</code> is, when not in the source directory
-C	use no configuration file
-D name=value	override a setting; -A name=value sets an HTML context value
-t TAG	define a tag for <code>only</code> directives
-n	nitpicky: warn about every unresolved reference
-W	turn warnings into errors; <code>--keep-going</code> reports all before failing
-w FILE	also write warnings to a file
-q, -Q, -v	quieter, silent, more verbose

There is no Makefile or `make.bat`: `guidedog build` is the make step on every platform. `guidedog clean` removes the build directory's contents, as `make clean` does.

Project layout

`guidedog quickstart` writes, with `--sep` adding the `source/` and `build/` split:

```

docs/                                docs/
  conf.toml                          source/
  index.rst                          conf.toml
  .gitignore                         index.rst
  _static/                           _static/
    guidedog.css                     guidedog.css
    guidedog.js                     guidedog.js
  _templates/                        _templates/
    layout.html                     layout.html
    book.typ                         book.typ
  _build/                            .gitignore
                                   build/

```

`index.rst` is Sphinx's blank root document: a comment, a title, one paragraph pointing to the reStructuredText documentation, and an empty root `toctree`

with `:maxdepth: 2` and `:caption: Contents:`. Prompts ask what `sphinx-quickstart` asks; `-q` takes defaults and flags. `.gitignore` ignores the build directory and Guidedog's caches.

The default templates are copied into the project, as editable files: `layout.html` and `guidedog.css/guidedog.js` for HTML, and `book.typ` for the PDF book. A build uses the project's copy when it exists and the built-in one otherwise, so deleting a file restores the default.

conf.toml

`conf.toml` uses the names of `conf.py` settings, at the top level, with TOML values, so a Sphinx user recognizes every line:

```
project = "Demo"
copyright = "2026, Ada"
author = "Ada"
version = "0.1"
release = "0.1.0"
language = "en"
extensions = []
templates_path = ["_templates"]
exclude_patterns = ["_build", "Thumbs.db", ".DS_Store"]
html_theme = "guidedog"
html_static_path = ["_static"]
```

Dictionaries are TOML tables (`[html_theme_options]`) and lists of tuples become arrays of inline tables (`pdf_documents = [{root = "index", file = "demo.pdf", title = "Demo"}]`). Every supported setting is validated by name and type; an unknown setting is an error with the closest known name. Supported settings, grouped as in the Sphinx documentation:

- **Project:** `project`, `author`, `copyright`, `version`, `release`, `today`, `today_fmt`.
- **General:** `root_doc` (`master_doc`), `source_suffix`, `source_encoding` (UTF-8 only), `exclude_patterns`, `include_patterns`, `include_roots`, `templates_path`, `extensions`, `rst_prolog`, `rst_epilog`, `default_role`, `primary_domain`, `highlight_language`, `pygments_style`, `numfig`, `numfig_format`, `numfig_secnum_depth`, `math_number_all`, `math_eqref_format`, `math_numfig`, `nitpicky`, `nitpick_ignore`, `nitpick_ignore_regex`, `show_authors`, `add_function_parentheses`, `add_module_names`, `toc_object_entries`, `trim_footnote_reference_space`, `smartquotes`, `smartquotes_action`, `smartquotes_excludes`, `keep_warnings`, `suppress_warnings`, `needs_sphinx` (ignored).
- **Language:** `language`, `locale_dirs`, `today_fmt` is localized.
- **HTML:** `html_theme`, `html_theme_options`, `html_title`, `html_short_title`, `html_logo`, `html_favicon`, `html_static_path`, `html_extra_path`, `html_css_files`, `html_js_files`, `html_last_updated_fmt`, `html_permalinks`, `html_permalinks_icon`, `html_use_index`, `html_split_index`, `html_copy_source`, `html_show_sourcelink`, `html_sourcelink_suffix`, `html_show_copyright`, `html_show_sphinx` (shows Guidedog), `html_baseurl`, `html_file_suffix`, `html_link_suffix`, `html_secnumber_suffix`, `html_codeblock_linenos_style`, `html_context`, `html_additional_pages`.
- **PDF** (the LaTeX settings' counterpart, typeset by Typst): `pdf_documents`, `pdf_paper_size`, `pdf_logo`, `pdf_toplevel_sectioning`, `pdf_show_urls`,

`pdf_preamble` (a Typst file for package imports and show rules), `pdf_packages` (download or offline), `pdf_fonts` (system or embedded), `pdf_font_paths`.

- **Built-in extensions:** `extensions` accepts `sphinx.ext.mathjax`, `sphinx.ext.imgmath`, `sphinx.ext.todo`, `sphinx.ext.githubpages`, `sphinx.ext.ifconfig` (with literal expressions), `sphinx.ext.extlinks` (with its dictionary), `sphinx.ext.autosectionlabel` (with `autosectionlabel_prefix_document` and `autosectionlabel_maxdepth`), `sphinx.ext.graphviz` (with `graphviz_output_format` and `graphviz_dot_args`), `sphinx.ext.intersphinx` (see below), and `sphinx.ext.autodoc` (with Sphinx’s `autodoc_*` settings, `autoclass_content`, and `autodoc_source_paths`, the folders its Python source is read from), which Guidedog implements natively. Any other extension is an error that names it and says it needs the future plugin system.

`conf.py` is a Python program, so not all of it can become TOML. `guidedog migrate` reads it without running it: literal assignments, names bound earlier, string concatenation and formatting, f-strings, and the current year become settings; `latex_documents` becomes `pdf_documents`. Everything else (`setup(app)`, conditional blocks, environment lookups, imports) is reported with its line, and the lines behind settings Guidedog could not convert stay in `conf.toml` as comments. The converted file is loaded before it is written, so a migration never writes a configuration that does not load.

Builders

Builder	Output
<code>html</code>	one page per document; <code>genindex.html</code>, <code>search.html</code>, <code>searchindex.js</code>, <code>py-modindex.html</code> when modules exist, <code>objects.inv</code>, <code>_static/</code>, <code>_sources/</code>, <code>_images/</code>, <code>_downloads/</code>
<code>dirhtml</code>	as <code>html</code> , with <code>docname/index.html</code> pages
<code>singlehtml</code>	the whole toctree on the root document’s page; each document’s anchors carry its key, so links, the index, and search reach the right one
<code>pdf</code>	one Typst-typeset PDF per <code>pdf_documents</code> entry, in <code>_build/pdf</code>
<code>text</code>	plain text per document, in <code>_build/text</code>
<code>gettext</code>	message templates (<code>.pot</code>) per document or per folder
<code>dummy</code>	read and link every document, reporting what linking reports; no output

The PDF book follows the toctree from its root document: each top-level toctree entry is a chapter, deeper entries nest, and each document’s sections are shifted by its toctree depth, as the Sphinx LaTeX builder does.

The build environment and incremental builds

A build has four phases. **Read**: parse the outdated documents and record, per document, its title, section tree, toctrees, labels, targets, terms, index entries, domain objects, numbered items, and dependencies (included files, images, downloads, and literal includes). **Consistency**: warn about documents in no toctree (unless marked `:orphan:`), duplicate labels, and toctree cycles. **Resolve**: build the global toctree, number sections and figures, and resolve every cross-reference. **Write**: render each page whose inputs changed.

A document is outdated when its source or any dependency changed, when it is new, or after `-E`. Its tree is kept as an object below `.doctrees/objects`, in the folders of its docname and named by the start of the digest of its bytes (`objects/usage/install.<digest>.gdo`), and checked against the full digest recorded when it was written before it is used. A document's search words are kept the same way below `.doctrees/search`. Because a file of that name never changes, the environment that names it can use it whatever a later build that failed read; files no environment names are removed when a build commits.

An object is only a cache of its source. When one a page needs is missing, differs from the digest recorded for it (damaged on disk, or another document's), or cannot be loaded, the build does not fail: it reads that document again from its source, writes its object again, and uses the new one, with a warning naming what was wrong with the cache (`build.object.stale`, "CACHED DOCUMENT READ AGAIN"). What the document declares was resolved from the first read, so the source must still be the one read then; only a document whose source changed during the build, or that cannot be read again, fails it. No `-E` is needed to recover. (Earlier, any such object failed every build until `-E`.)

Writing works in output units: a document's page, the files the site shares, the search index, a book. A unit is kept when the digest of its inputs is unchanged: its source and included files, the whole of `conf.toml` and `-D`, the builder, the `-t` tags, `-n`, the navigation, the templates (every file below `templates_path`), translations, inventories, and the executable. What a page's references resolve to is not guessed from the tables: the link pass records each lookup it makes (a label, a term, an object by name or by the end of its name, a document, a file, an image) and the next build looks each one up again. A kept unit reports the diagnostics it gave when it was written, so `-w` fails or passes the same way whether or not anything was rewritten. `-a` writes every unit. The dummy builder links every document without writing.

Publication

This section supersedes the implemented decision of GDS 0002, "Publication is a host transaction", for project builds.

A build publishes its output folder, the manifest (`.doctrees/<builder>.outputs.json`, each unit's digest, files, assets, lookups, and diagnostics), and the environment (`.doctrees/environment.json`) together, or none of them, whatever fails and wherever the process stops. It publishes only when it did all its work and `-w` does not fail it: `-w` is decided before anything is published. A failed build, for any reason, leaves the previous output, manifest, and environment exactly as they were, so the next build reads and writes that work again and reports the same diagnostics. Sphinx writes each page as it finishes it,

so a `-W` failure there leaves a site part old and part new; Guidedog deliberately does not.

The problem with moving files. An earlier design (implemented until 2026-09-30) staged the changed files in a folder beside the output and, once decided, moved them into the output one by one and removed the obsolete files after. The second external review showed that this cannot be made whole-output atomic: when a file becomes a folder (`html_extra_path` holding `extras/z`, then `extras/z/nested`), the move of `z/nested` fails because the old `z` is still there; the changed page had already been moved, the environment was still old, the summary said the previous output was unchanged, and every later build failed again replaying the journal. Any failure in the middle of the moves (permissions, a full disk, a file a program holds open) had the same effect. The fix is not a better order of moves, but not moving files into the live output at all.

Decision: generation folders with a switch. The build writes the next output as a whole folder beside the live one (`_build/.html.next`), and publishing it is switching the two folders.

- While the build runs, every file it changes is written to the next folder (through a temporary file and a rename); a file whose bytes are already published is only recorded. Nothing in the live output changes.
- When the build succeeded, the manifest and the environment are written beside their destinations (`<builder>.outputs.json.next`, `<builder>.environment.json.next`, each only when its bytes differ), and the record each will replace is kept as a hard link (`.previous`), to put back if the commit is undone.
- The next folder is completed into the whole next output (`complete_next`): every file of the live output that is not obsolete (made by the previous generation and not by this one) is hard-linked into it, or copied where the file system cannot link (FAT, some network shares); files Guidedog did not make, such as a `CNAME`, are kept. Because the next output is a tree of its own, a file that becomes a folder, or a folder that becomes a file, is like any other change. Only a file Guidedog did not make that stands where the next output needs a folder, or below a path where it needs a file, stops the build: `A FILE IN THE OUTPUT IS IN THE WAY`, naming it, with the hint to move it or run `guidedog clean`. Nothing is published.
- Everything the journal will name is made durable in one pass (below).
- The journal (`.doctrees/<builder>.commit.json`, version 2) is saved through a flushed temporary file and a rename, and its folder flushed. It names the output folder, the next folder and its identity (device and inode), where the previous output goes in a two-rename switch (`.html.old`), and the records with their `.previous` links. Until it exists nothing published has changed; once it exists the commit is decided.
- The switch: where the system can, the output folder and the next are exchanged in one step (`renameat2` with `RENAME_EXCHANGE` on Linux, `renamex_np` with `RENAME_SWAP` on macOS; `host.exchange`), so a reader of the output folder, `guidedog serve` or a web server, sees the old tree or the new one and never neither. Elsewhere (Windows, file systems without the operation) it is two renames, output to `.html.old`, next to output. Then each record is renamed

over its destination, the renamed folders are flushed, and the journal is removed. The `.previous` links go last.

Recovery. Each step can be repeated and each can be undone. Where the new output is is told by the identity recorded in the journal, not by names, so a switch stopped anywhere, even between an exchange and the step after it, is recognised; without identities (where the system gives no inode) there is no exchange, and the names tell. No step guesses: a path whose state cannot be read (for want of memory, or a permission) is a failed step, never “not there”.

- A step that fails is undone in this build: the renamed records are renamed back beside their destinations and their `.previous` links put back, the folders switched back, the journal removed. The problem is `THE OUTPUT COULD NOT BE PUBLISHED`, saying what failed and that the previous output and records are as they were, with the hint to close any program holding a file of the output open, check write permission and free disk space, or run `guidedog clean`. The summary says the previous output is unchanged, which is then true.
- When undoing fails too, the journal stays and the problem is `PUBLICATION NOT FINISHED`; the summary says the publication is not finished and that the next build finishes or undoes it, never that the previous output is unchanged (`Build_Result.unsettled`).
- A build begins by settling every journal it finds, of any builder, before it reads anything: it finishes the commit; if a step cannot be done, it undoes it, warns (`INTERRUPTED PUBLICATION UNDONE`), and goes on. A journal whose new output is gone is undone the same way. A journal that cannot be read (damaged, or written by an older Guidedog, such as the version 1 journal the review’s build left) is discarded with a warning (`UNREADABLE PUBLICATION JOURNAL DISCARDED`), and the builder’s manifest keeps its files but loses its digests, so the build writes every output again and compares it with the file in place. So no build is stuck behind a journal it cannot complete; only a commit that can be neither finished nor undone (a file system that refuses both) stops the build, with the paths and `guidedog clean` in the hint.

The records must not move with the output. Default builder folders keep their records in the build folder’s `.doctrees`, outside the published tree. A custom output keeps its records in `./<output-name>.guidedog/.doctrees`, a stable sibling. This matters for `guidedog build -b html SOURCEDIR OUTPUTDIR`, where putting records inside `OUTPUTDIR` would move the journal during the directory switch. Each custom target has its own manifest, so existing filenames in a different output cannot stand in for that target’s published contents. Journal paths are absolute and resolve existing ancestors before naming a not-yet-created output, so recovery is independent of the next process’s working directory and filesystem aliases. An output cannot contain the source directory. `lib/project/build_paths_test.odin` checks fresh and unchanged direct outputs, interrupted commits, destination isolation, and that source-layout refusal.

Cost: the spare output. Linking every file of the site into a fresh folder, and removing the replaced tree afterwards, costs two system calls per file for every publishing build. On an 1,800-document project (3,692 entries; Linux, ZFS, 32 cores), this made a one-page rebuild 0.62 s instead of 0.43 s, then 0.55 s with folder

listings that open no entry (`host.list_folder`, from `readdir`'s names, types and inodes). So the replaced output is not removed: after an exchange it is already in the next folder, and after two renames it is moved there once the journal is gone. It is the spare output the next build writes into and completes, and it shares every unchanged file with the live output (they are hard links to the same files). Completing it compares the two trees folder by folder, by name, type, and inode from the listings: a name that is the same file as the live one is kept, one that differs is replaced by a link, one the live output does not have (and this build did not write) is removed. Nothing in the spare output is trusted: a stray file, a folder where a file goes, a file where a folder goes, or a file under a kept name that is not the live file is replaced or removed (`the_spare_output_never_leaks_into_the_published_one`). A producer that writes its own file (Typst) is given a path with nothing at it, so it can never write through a link into a published file; a folder a user made read-only in the output, once it is the spare output, is made writable again, since the spare is Guidedog's own. With the spare output, the same one-page rebuild is 0.43 s, as before the change (five runs: 0.43 s each after the first two, which fill the spare output once); a clean build and an unchanged build are unchanged (2.6 to 3.1 s and 0.33 s). The cost of a publication is listing the folders plus a link per changed file.

Durability, efficiently. A file is durable only once flushed (`fsync`), and a flush waits for storage: flushing each output as it was written made a clean build of CPython's documentation spend about 91% of its time in 1,742 flushes on ZFS. The journal needs only that what it names is durable before it is saved, and that its renames are durable before it is removed. So files are written without flushing (`host.write_unsynced`, still through a temporary file and a rename): the files this build wrote, the files copied where they could not be linked, the `.next` and `.previous` records, and the objects and search words a build reads. Just before the journal is saved, `host.sync_all` flushes all of them with a small pool of threads (`SYNC_WORKERS`, 8), then each folder of the next output whose names changed, up to the build folder, once (`host.folders_of`). A linked file is the published file itself, already durable. After the switch and the record renames, the build folder and the records' folder are flushed, then the journal is removed. Windows flushes no folder (it keeps names durable itself). A crash before the journal is durable leaves the previous output; after, the next build finishes the commit from files that are all on storage. A build that changes nothing (no file written, none obsolete, both records as they would be written) saves no journal and switches nothing, so an unchanged rebuild leaves the output folder, `environment.json`, and `<builder>.outputs.json` as they were, times and inodes included (`an_unchanged_build_rewrites_no_records`); a build that changes only records leaves the output folder in place.

Memory. Every allocation a commit makes before it is decided is checked; one the system refuses is `OUT OF MEMORY` (a problem that allocates nothing), and nothing is published. After the decision, the steps allocate only paths' C strings and state reads in stack buffers, and a refused one is a failed step, undone as above.

Alternatives rejected.

- *Preflight planning of file moves.* Compute every move and removal, including removing a conflicting file or folder before creating a path, check each

step's preconditions before the decision, and make each step idempotent and reversible. It keeps the live output being edited in place, so a reader sees a mixture while the moves run, and undoing a removal means keeping every removed file aside until the end: the same links a generation folder makes, without its single switch. Preconditions checked before the decision can still fail after it (a program opens a file, the disk fills). Rejected for the generation switch, which has one step to undo.

- *A symbolic link as the output folder*, switched with one rename. It makes `_build/html` a link, which some servers, archivers and Windows (without developer mode) handle badly, and changes what users see. Rejected.
- *Copying unchanged files* instead of linking. Correct, but it reads and writes the whole site for every change. Kept only as the fallback where linking fails.
- *Removing the replaced output after each switch*. Simple, but it costs a link and a removal per file per build (measured above). Replaced by the spare output.

Limits. The output folder is switched by renames in its parent, so it cannot be a mount point (a bind-mounted `_build/html` in a container); mount the build folder instead. The next folder and the previous one are siblings of the output folder, hidden by their leading dot (`.html.next`, `.html.old`); `guidedog clean` removes them with the rest of the build folder. With `-o`, they are beside the named folder.

Tests. `lib/project/commit_test.odin`: the review's case, a page change with a file that becomes a folder, and back (`a_file_that_becomes_a_folder_is_published`); a failure injected at every step of the commit, with the folders exchanged and with two renames, leaves the previous output and records byte for byte, and the next build publishes (`a_failed_commit_step_is_undone`); a stop at every step, as when the machine goes down, is finished by the next build even when that build fails by `-W` (`an_interrupted_commit_is_finished_by_the_next_build`); real failures of the system: a build folder made read-only just before the switch (`a_read_only_build_folder_fails_the_commit_cleanly`) and a records folder made read-only after it, so that undoing fails too (`a_commit_that_cannot_be_undone_is_finished_by_the_next_build`); an unreadable journal (`an_unreadable_journal_is_discarded`); a user's file in the way; and the spare output never leaking. `tests/publication_test.odin` runs the review's case and the real failures through the `guidedog` command, checking the summary's words.

Cross-document resolution happens between reading and writing, in the host: a pass over each document replaces toctrees with navigation lists and cross-references with links, using the environment. Documents are still read and rendered by the allocation-free engine.

Project surface

The builder is a library, `lib/project`, and `lib/cli` is only its command line, so any Odin program can drive a build the way `guidedog` does. Its public surface is a handful of verbs over a loaded project and the types they take and return; everything else (discovery, reading, the environment and its cache, cross-references, templates, search, graphs, intersphinx, and the commit journal) lives in `#+private` files.

- **Projects:** `Project`, `Load_Options`, `load(dir, options, registry)`, `unload`, and `CONFIG_FILE`. `Config` is public with the types of its fields (`Pdf_Document`,

Flag_Or_Name, Intersphinx_Project, Toml_Value), since a caller may change the loaded configuration before a build, as `--typst` does.

- **Building:** `Builder_Kind` with `BUILDER_NAMES` and `builder_named` (the LaTeX builders' names map to `pdf`), `Build_Options`, `BUILD_BUDGET`, `build_limits` (for refusing options before a build), `build`, `Build_Result`, `release`, and `clean`.
- **Preview, translations, starting, migrating:** `Server` and `serve`; `Intl_Options`, `intl_update`, `intl_stat`, `Intl_File`, `Intl_Action`; `Quickstart_Options` and `quickstart`; `Migrate_Options`, `migrate`, `Migrate_Result`.
- **Paths:** `absolute` (against the working directory, without resolving a folder that does not exist yet) and `join`.

Ownership is explicit and uniform. `load` keeps the project in an arena of its own that `unload` frees, the text of a failed load's problem included, so `unload` is called either way. `build` allocates the build and its result in an arena of the result's own, which `release` frees; a build never changes the project (it works on a copy of the configuration), so one project may be built any number of times. `serve` answers each request in an arena it frees, and its rebuild callback releases the results of the builds it runs. The other verbs allocate their results in `context allocator`, which a caller on the heap makes an arena. `load`, `build`, `release`, `unload` leaves nothing live. Failure is a `host.Problem` with a non-zero exit, never a panic.

`lib/project/README.md` lists the surface with a complete program. The doc-comment test requires a contract comment on each exported procedure, and `tests/surface_test.odin` requires each exported name to appear in the cheat sheet, so the surface cannot grow without being advertised. The host services the builder uses are GDS 0003's host surface.

Directives and roles

Guidedoc's RST reader gains an extension interface: tables of directive and role handlers consulted after the standard ones. The Sphinx layer is its first user, and future plugins will be its next. Sphinx directives produce these nodes (added to the AST contract):

- **toctree** (all options: `maxdepth`, `caption`, `name`, `hidden`, `includehidden`, `numbered`, `titlesonly`, `glob`, `reversed`), `only`, `index` and the `:index:` role, `glossary`, `productionlist`, `centered`, `hlist`, `seealso`, `rubric`, `versionadded`, `versionchanged`, `deprecated`, `versionremoved`, `sectionauthor`, `moduleauthor`, `codeauthor`, `tabularcolumns`, `highlight`, `code-block`, `sourcecode`, `code` with Sphinx options (`linenos`, `lineno-start`, `emphasize-lines`, `caption`, `name`, `dedent`, `force`), `literalinclude` (every option), `math` with `label` and `nowrap`, `default-domain`, `default-role`, `todo` and `todolist`, `ifconfig`, and the figure and table numbering options.
- **Roles:** `doc`, `ref`, `numref`, `any`, `term`, `keyword`, `option`, `envvar`, `token`, `download`, `abbr`, `command`, `dfn`, `file`, `guilabel`, `kbd`, `mailheader`, `makevar`, `manpage`, `menuselection`, `mimetype`, `newsgroup`, `program`, `regexp`, `samp`, `pep`, `rfc`, `math`, `eq`, `math:numref`, and `extlinks` roles.
- **Domains:** `py` (module, currentmodule, function, data, exception, class, attribute, property, method, staticmethod, classmethod, decorator, decorator-method, type), `c`, `cpp`, `js`, `rst` (directive, directive:option, role), and `std`

(program, option, cmdoption, envvar, describe, object, confval, term, label), with their roles. Python signatures are parsed into name, parameters, defaults, annotations, and return type; C and C++ declarations are tokenized to find the declared name, and are shown as written.

Themes, templates, and pages

HTML pages are produced from `layout.html`, a Jinja template. Guidedog includes its own implementation of the Jinja language in Odin, `lib/jinja`: inheritance with `extends` and `block`, `include`, `import`, `macros` and `call`, `set`, `with`, `filter`, loops with the `loop` variable, conditionals, whitespace control, autoescaping, and Jinja’s built-in filters, tests, and globals. Page variables include `project`, `title`, `body`, `toc` (the sidebar navigation), `outline`, `prev`, `next`, `path_to_root`, `css_files`, `js_files`, `language`, `copyright`, `version`, `release`, `last_updated`, `sourcelink`, and the `html_context` values, with their types. `html_additional_pages` renders pages from templates alone, with the same variables, at the site’s root. The PDF book is `book.typ`, an ordinary Typst template that receives the book’s metadata and content.

The default `html_theme` is `guidedog: responsive`, readable without JavaScript, light and dark, with a sidebar toctree, an on-page outline, previous and next links, search, copyable code, and permalinks. `html_theme_options` sets its tokens. `alabaster`, `classic`, `sphinx_rtd_theme`, `furo`, and `pydata_sphinx_theme` are accepted and use `guidedog` with a note, so an existing `conf.toml` needs no edit to build.

Code highlighting

Sphinx highlights code with Pygments, a Python library. Guidedog has its own highlighter, written in Odin as a separate library (`lib/highlight`): 35 languages with Pygments’ aliases, including console sessions, tracebacks, reStructuredText, and Graphviz, and Pygments’ token classes, so stylesheets written for Pygments apply. `highlight_language` (with Sphinx’s `default`, which falls back to plain text when code is not Python) and `pygments_style` work as in Sphinx, with every built-in Pygments style; `pygments_dark_style` styles dark pages. Unset, they are the theme’s: `default` and `github-dark`. Renderers call it through `Render_Options.highlighter`; the book keeps Typst’s highlighting. No Python runs at any point: not in builds, not in highlighting.

Reading reStructuredText completely

The `reStructuredText` reader implements the constructs, directives, and roles listed in its coverage ledgers, including smart quotes, PEP and RFC references, trimmed footnote reference space, tab width, the document title, and bibliographic fields. The current ledger has no open entries; the API’s conservative conformance descriptor remains `partial`. A finite corpus cannot prove equivalence for every input. Conformance is measured: a differential harness compares Guidedog’s document tree with Docutils 0.22’s on Docutils’ own functional test inputs and on the `reStructuredText` specification documents.

Markdown

Markdown is first-class in Guidedoc’s `CommonMark` reader; there is no separate reader and no plugin. Project `.md` documents support the native MyST syntax listed below. The pinned `myst-parser` 0.16.1 fixtures provide

comparison data, with recorded differences; they do not promise that every MyST project builds unchanged. Core syntax includes: directive fences (`{name}`) with options, roles (`{role}` followed by backquoted content), targets (`(label)=`), comments, block breaks, front matter, footnotes, tables, and cross-references. Its optional syntax is enabled as in Sphinx, with `myst_enable_extensions: amsmath, colon_fence, deflist, dollarmath, fieldlist, html_admonition, html_image, linkify, replacements, smartquotes, substitution, and tasklist`, with their `myst_*` settings. `extensions = ["myst_parser"]` is accepted and needs nothing, because the support is built in. Directives and roles are the same implementations reStructuredText uses, plugged into the reader through a hook, so a Markdown page produces the same document tree, pages, and book as its reStructuredText equivalent. A strict mode keeps plain CommonMark conformance for `guidedog convert`. `myst-parser 0.16.1`'s test fixtures, used as data, measure the rest.

Graphs

`sphinx.ext.graphviz`'s markup (`graphviz`, `graph`, and `digraph`, with every option) and Markdown's `dot` fences draw graphs. Graphviz is linked into Guidedog through `graphdog`, an Odin library over Graphviz's C library that can also be built statically; no `dot` program runs. Each graph is laid out once per distinct source and cached as an SVG in `_images`; pages show it as an image and the book embeds the same SVG. A captioned graph is a numbered figure. A graph with an error stays code, with a warning at its line.

Typst

Typst files are not documents. They are the PDF layer: `_templates/book.typ` decides how the book looks, `pdf_preamble` adds rules and package imports, and `.. raw:: typst` (or a `{raw}` typst fence in Markdown) puts Typst markup into the book alone. Listing `.typ` in `source_suffix` is an error that says so.

The book is set with Typst's optimized (Knuth–Plass) line breaking, justified, with hyphenation in the document language and costs against widows, orphans, and runts. Guidedog ships Libertinus Sans (SIL Open Font License) beside the Libertinus Serif Typst embeds, so the book looks the same on every machine.

Intersphinx

`sphinx.ext.intersphinx` links references to other projects through their inventories (`objects.inv`), as Sphinx does. `intersphinx_mapping` is a table of names to a target URI and a location, or a list of locations; `""` stands for None, the `objects.inv` at the target URI. `intersphinx_cache_limit`, `intersphinx_timeout`, `intersphinx_disabled_reftypes` (default `std:doc`), and `intersphinx_resolve_self` work as in Sphinx. Inventories are fetched over HTTP(S) through `libcurl` (`lib/fetch`) and cached in `.doctrees/_intersphinx_cache_`; they are read and written by `lib/inventory`. Resolution follows Sphinx's: a reference the project cannot resolve is looked up by the object types its role names, then qualified by its scope; a target `name:target` looks only in that inventory; labels and terms match ignoring case; and the `:external:` roles resolve only through the inventories. Guidedog's own `objects.inv` lists what Sphinx's domains list, with their priorities.

Rationale and alternatives

Executing `conf.py` would require Python and would make builds depend on arbitrary code; TOML with the same names keeps the familiarity without the dependency, and `migrate` covers the literal settings real projects use. A Makefile adds nothing a command line cannot do and does not run natively on Windows. Reproducing Sphinx's HTML byte for byte would tie Guidedog to Sphinx's templates; stable paths and anchors are what keep links and bookmarks working. Implementing Sphinx features natively rather than through a plugin API first lets that API be designed from real use.

Security and operational considerations

Sources are read only under the source directory; includes, literal includes, graph files, and images follow the confinement rules of GDS 0002. Sphinx reads any path, and projects such as CPython include files from beside the source folder; a project opts into that with `include_roots`, a list of folders relative to the source folder, and nothing else outside it is read. An image from an include root is kept in `_images` by its name. Raw content follows the configured policy. A document, image, template, or static file that is a link must lead into those folders too; the path is resolved before it is compared (on Windows a path through a link is refused). A build trusts its project by default; `--untrusted` leaves out raw content and unsafe URLs with a warning and fetches nothing. A failed build publishes nothing; see the build environment above.

`sphinx.ext.autodoc` reads Python source and never executes it: Sphinx imports the project's package, which runs its code during the build; Guidedog parses the files with tree-sitter (native/tree-sitter builds pinned, checksummed releases) and emulates what the import would have produced. Python files are read only below `autodoc_source_paths`, with the same link and confinement rules as includes, and each one read is a dependency of the page. What only running the code could reveal (computed values, objects made by decorators from other packages, compiled modules) is reported by name, never guessed silently.

Backwards compatibility

Projects created by the earlier `guidedog init` are not supported by the new builder; `guidedog migrate` also converts a `guidedog.toml`. Existing Sphinx projects need `guidedog migrate` once to produce `conf.toml`; their sources are unchanged.

Acceptance criteria and implementation status

- `quickstart` reproduces `sphinx-quickstart`'s layout and blank index.
- A differential harness against Sphinx 9.1 on sample projects and Sphinx's own test roots compares page sets, toctree navigation, anchors, link targets, numbering, and warnings.
- A differential harness against Docutils 0.22 compares document trees on Docutils' functional inputs; both RST ledgers have no open gaps.
- Fresh and incremental builds produce identical output; editing one document rewrites exactly the pages that depend on it.

The supported project surface is implemented. Evidence is retained in `docs/manual/evidence/beta-review-20261011.md` and the reader ledgers.

- `quickstart`, the supported builders, `clean`, `serve`, `migrate`, and native gettext catalog commands operate on project folders.
- The Docutils comparison matches 98 of 98 corpus sources. CommonMark passes all 652 pinned examples. MyST matches 215 of 218 reader fixtures and 8 of 10 Sphinx fixtures; the recorded differences remain explicit.
- The Sphinx 9.1 differential harness exists in `tools/sphinxdiff` and runs against Sphinx's own roots. C++ identifiers, inventories, and grouped info fields are implemented. The fixture notes explain the remaining differences.
- Publication switches complete generations with recovery and rollback tests. Unchanged outputs are reused; `-w` is decided before publication.
- Generation cleanup uses a checked iterative postorder work list. Files and symbolic links are unlinked without traversing their targets. `clean` refuses a symlink root. Tests cover ordinary files, outside links, and refusal of each cleanup allocation; deep trees do not require recursive stack growth.
- Project allocations are charged before allocation. Load failures return a memory problem and remain safe to unload. Native compiler allocations have a separate operating-system boundary; GDS 0004 specifies it.
- The preview fingerprints watched paths and file metadata. Deletions, edits below a future timestamp, shared extra files, and removed roots are covered by regression tests. Generated outputs and hidden runtime state are excluded.
- The manual exercises native PO/POT translation in six languages. The API project exercises the Odin domain without executing the documented code.
- CPython, Django, and Flask clean and unchanged HTML/PDF runs are repeated remotely. Their extension warnings remain measured limitations.
- The embedded and external Typst backends share the reproducible-date contract. Epoch range checks precede nanosecond conversion; explicit offsets retain time of day and use checked addition. Date tests run against both backends.

The implementation status concerns this supported contract. Python extension execution, additional Sphinx builders, a future plugin ABI, and complete dynamic autodoc equivalence are outside this record's beta scope. They are not inferred from a successful build.

Open questions

The plugin interface is left to a later record.

Review history

29 September 2026: created after the user corrected the direction: Guidedog is a Sphinx replacement with a project layout, `conf.toml`, builders, and incremental builds, with complete reStructuredText, and no Makefile.

29 September 2026, review of the build pipeline: outputs are published as one generation with a manifest; pages record their lookups and replay their diagnostics; cached objects are stored by docname and checked by digest; singlehtml namespaces anchors by document; links are confined like paths; `--untrusted`; search indexes every section.

30 September 2026, direction: Guidedog is not a drop-in replacement. The earlier wording aspired to everything Sphinx does. On 11 October the maintainer

clarified the target: an independent modern documentation system, designed idiomatically in Odin, with no Python execution and no goal of complete Sphinx compatibility. Sphinx is inspiration, not the specification. Comparisons measure useful authoring behavior and differences.

30 September 2026, publication: the output, the manifest, and the environment are published together by a journaled commit that the next build finishes after a crash; `-w` is decided before publishing, so a build it fails changes nothing; objects and search words are named by digest and swept after a commit. This supersedes the implemented publication decision of GDS 0002.

30 September 2026, final audit of the external review: a build publishes only what changed (an unchanged rebuild writes no record and no journal); a commit flushes what it publishes in one concurrent pass before its journal instead of one flush per file (a clean build of 400 documents on ZFS: about 9 s to 0.45 s); a damaged or unusable cached object is read again from its source instead of failing every build until `-E`; and the build summary says whether the output was published.

30 September 2026, API review: `lib/project` exports only the project verbs and the types they take (“Project surface”); its implementation files are private, every exported procedure states its contract, and its cheat sheet lists the surface.

References

- GDS 0002, Guidedog: CLI and documentation toolchain.
- GDS 0003, Guidedoc: Odin conversion engine.
- [Sphinx documentation](#) (9.1).
- [reStructuredText specification](#).

Lifecycle event

2026-10-11: prediscussion → discussion. Assigned a permanent number and opened for discussion.

Lifecycle event

2026-10-11: discussion → accepted. Maintainer-requested implementation-status reconciliation; current supported contract reviewed and deferred scope stated explicitly.

Lifecycle event

2026-10-11: accepted → committed. Current supported design implemented; beta review fixes and limitations are recorded. Native library and integration suites with leak checks; sanitizer and embedded-backend validation; docs/manual/evidence/beta-review-20261011.md. Deferred features are not claimed as implemented.

Historical provenance

GDS inherits its process from the Zen Discussion Process. The original record has been ported to RST for reading here. Its historical state and claims concern `zenfmt`, not Guidedog. Original Typst sources remain in Git history.

The Zen Discussion Process

Abstract

The `zenfmt` monorepo needs a durable decision-record process for architectural, format, performance, and documentation changes across the conversion library and the `zenfmt` command-line tool. A document converter accumulates decisions that are invisible in the code and expensive to relearn: which parts of a source format are deliberately dropped, why a construct maps to one Markdown spelling rather than another, and what the plugin contract promises a reader that the engine will never do on its behalf. The project uses Zen Discussions, or ZDS, as RFC/RFD-style Typst documents that support structured reasoning, long-lived references, high-quality PDF output, and an HTML discussion website.

This memo defines the lifecycle of a ZDS, the placeholder numbering workflow, the authoring expectations, and the Typst project layout.

Introduction

A Zen Discussion is a Typst document stored in git under `docs/zds/records`. Each discussion is part design memo, part review artifact, and part historical record. The structure is intentionally close to IETF RFCs and Oxide RFDs because those formats force explicit scope, status, rationale, alternatives, and operational considerations instead of relying on implicit context.

ZDS is used for topics such as:

- the document intermediate representation and its compatibility rules
- the plugin contract: what a reader may assume, what a writer must handle
- the mapping from a specific source format to the IR, and the constructs that mapping deliberately discards
- performance and allocation budgets, and the benchmarks that defend them
- security decisions, notably the handling of untrusted archive and markup input
- contributor and documentation process decisions

Active design reasoning, trade-offs, and decisions belong in ZDS. The Typst book under `docs/book/`, when it is written, remains the descriptive manual of the current system; a ZDS records why the system became that way.

Why a converter needs written records

Every format zenfmt reads is a lossy projection onto one shared IR, and every format it writes is a lossy projection back out. The interesting engineering is almost entirely in those two projections, and almost none of it is legible from the code alone. A reader that skips a construct and a reader that has not yet implemented that construct look identical in a diff. Only a record distinguishes them.

This is why a mapping decision — even a small one, such as rendering a single-cell table as a paragraph — belongs in a numbered document with the alternatives that lost.

Terminology and Scope

- **ZDS**: a Zen Discussion document
- **placeholder draft**: a local Typst draft using the placeholder number `xxxxx`
- **assigned ZDS**: an accepted discussion with a permanent four-digit number
- **registry**: the Typst metadata list in `docs/zds/registry.typ` used by the index and bundle entry point
- **bundle**: the experimental Typst export target that emits the HTML index, HTML discussion pages, and per-ZDS PDFs from one entry point

This memo defines the repository workflow for ZDS. It does not define future CI automation in full detail, but it does define the expected behavior of that automation.

Discussion Lifecycle

New discussions begin as placeholder drafts.

- Contributors `copy docs/zds/template/rfc-template.typ` to `docs/zds/records/XXXXX-<slug>.typ`.
- The document uses the placeholder number `xxxxx` while the author iterates.
- The registry-driven Zen Discussions index includes placeholder drafts after assigned discussions.
- When the draft is ready, maintainers assign the next permanent ZDS number and rename the file to `NNNN-<slug>.typ`.
- The registry is updated with the document's area, status, summary, source path, HTML path, and PDF path.

This keeps cloning and local authoring simple. Contributors do not need a global lock or remote numbering check just to start writing.

Local Workflow Diagram

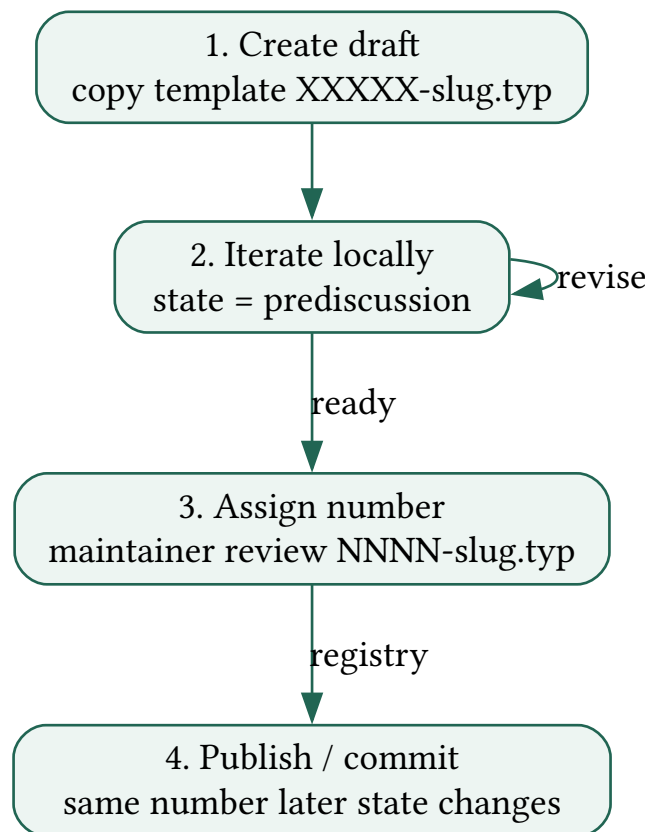


Fig. 9 The record lifecycle as four numbered steps: create a draft with a placeholder number and the prediscussion state; iterate locally, revising until it is ready; assign a permanent number and register it, moving to discussion and open for review; and finally publish and commit.

States

ZDS documents can move through these states:

- `prediscussion`: local draft or early working document
- `discussion`: open for review and feedback
- `accepted`: accepted as the intended direction, not yet fully implemented
- `published`: accepted into the repository as the current design record
- `committed`: fully implemented and now descriptive of the current system
- `abandoned`: deliberately closed without adoption

State Transition Diagram

Solid edges are the forward path of a discussion; dashed edges close a document without adoption. The filled dot is the moment a contributor copies the template.

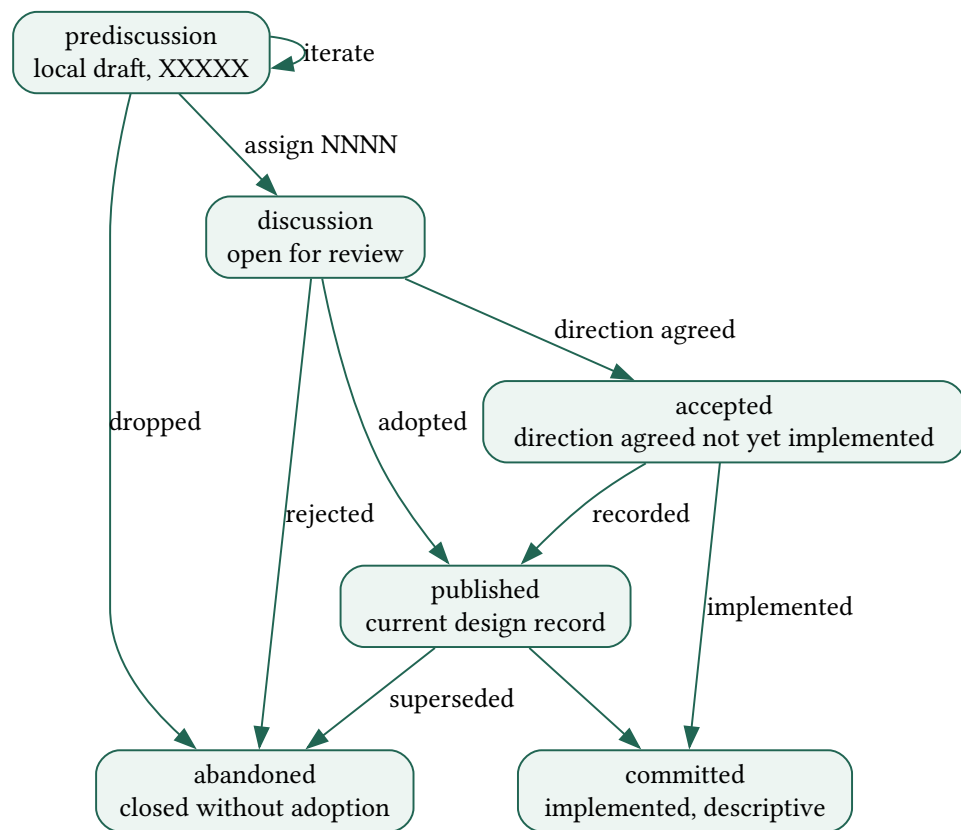


Fig. 10 The lifecycle states as a chain with the transition that leads to each: prediscussion (a local draft with a placeholder number) becomes discussion once a number is assigned, then accepted once the direction is agreed, then published as the current design record, then committed once implemented and descriptive of the system. From any state before committed a record may instead become abandoned — dropped, rejected, or superseded.

Transition Detail

Transition	Actor	What happens
start → prediscussion	contributor	Copy <code>template/rfc-template.typ</code> to <code>records/XXXXX-</code> and <code><slug>.typ</code> iterate locally under the placeholder number.
prediscussion → discussion	maintainer	Run <code>zig build zds-promote: -- <slug></code> it assigns the next four-digit number, renames the file to <code>NNNN-<slug>.typ</code> , rewrites the meta-data for discussion, and appends the <code>registry.typ</code> and <code>bundle.typ</code> entries.

Transition	Actor	What happens
discussion → accepted	maintainers	Review converges on the proposed direction; implementation has not landed yet. The registry state changes, nothing is renamed.
discussion → published	maintainers	The document itself is the deliverable (process memos, assessments, records) and is adopted as the current design record.
accepted → published	maintainers	The accepted direction is written up as the repository's current design record while implementation continues.
accepted / published → committed	maintainers	The described system is fully implemented; the ZDS is now descriptive of the current system rather than a proposal.
any pre-committed state → abandoned	maintainers	The discussion is deliberately closed without adoption: dropped while drafting, rejected in review, or superseded by a later ZDS. The number is never reused.

Authoring Rules

ZDS documents SHOULD follow the RFC-style template in `docs/zds/template/rfc-template.typ`.

Each ZDS is expected to include, at minimum:

- an abstract or decision summary
- a clear statement of scope and goals
- the main design or proposal
- considerations for security, operations, or workflow where relevant
- alternatives and unresolved questions
- Keep one ZDS per file under `docs/zds/records`.
- Store document metadata in the `#let zds-*` assignments at the top of the file.

- Update `docs/zds/registry.typ` whenever a ZDS is added, renumbered, renamed, or changes lifecycle state.
- Build through `zig build` once Zig is installed; Typst remains the document compiler.

The ZDS Typst layer is allowed to use external Typst packages for diagrams, charts, richer tables, or specialized layout when those packages improve the clarity of the PDF output. The repository should prefer existing Typst packages over building a custom extension framework here.

Format Records

A ZDS that adds or changes a format plugin carries three sections beyond the template, because these are the parts of a converter that reviewers cannot reconstruct from the diff:

- **Mapping table:** every source construct the plugin recognizes, and the IR node it produces. One row per construct.
- **Deliberate omissions:** source constructs the plugin sees and drops on purpose, each with the reason. This is the section that distinguishes a decision from a gap, and it is the section reviewers should read first.
- **Round-trip expectations:** what survives a conversion to the target format and back, and what does not. A converter that claims fidelity it does not have is worse than one that documents its losses.

Two rules follow from the shared IR. A change to `Block`, `Inline`, or their tags affects every plugin at once and requires its own ZDS. A change confined to one plugin's mapping does not, and may be recorded as an amendment to that plugin's existing record.

Typst Project Layout

The ZDS tree is a first-class Typst project:

- `docs/zds/records/` contains standalone ZDS source files.
- `docs/zds/template/rfc-template.typ` is the starting point for new ZDS drafts.
- `docs/zds/registry.typ` is the metadata registry used by the index and bundle.
- `docs/zds/index.typ` renders the PDF/HTML index from the registry.
- `docs/zds/bundle.typ` uses Typst bundle export to emit `index.html`, one HTML page per ZDS, and one PDF per ZDS.
- `docs/shared/zds.typ` and `docs/shared/theme.typ` provide the shared document frame, styling, and index components.
- `docs/book/` and `docs/book.typ` hold the skeleton of the future zenfmt book. The theme, callouts, and figure helpers are in place; the chapters are not written yet, and no build step compiles them.

Typst bundle export and HTML export are experimental in Typst 0.15. They are useful for the ZDS website, but the repository should still keep standalone PDF compilation for each ZDS as the stable archival path.

Build Integration

The root `build.zig` owns the ZDS build steps. Records are discovered by scanning `docs/zds/records`, so adding a record never requires a build-file edit:

- `zig build zds` compiles every numbered record to `docs/build/zds-NNNN-<slug>.pdf`.
- `zig build zds -Dzds=<number-or-slug>` compiles a single record; the number may be unpadding (`-Dzds=2`), and a placeholder draft can be selected by its slug for proofreading before promotion.
- `zig build zds-index` compiles the registry-driven index to `docs/build/zds-index.pdf`.
- `zig build zds-site` compiles the experimental HTML bundle to `docs/build/zds-site/`.

Lifecycle management is owned by `tools/zds.zig`:

- `zig build zds-list` prints registry entries and placeholder drafts, and warns when a record file and the registry disagree.
- `zig build zds-new -- <slug>` creates `records/XXXXX-<slug>.typ` from the template with today's date.
- `zig build zds-promote -- <slug>` performs the `prediscussion` → `discussion` transition described above.

Direct Typst commands are useful while editing:

```
typst compile --root docs docs/zds/records/0001-zds-process.typ \
docs/build/zds-0001-zds-process.pdf
typst compile --features html,bundle --root docs --format bundle \
docs/zds/bundle.typ docs/build/zds-site
```

Number Assignment and CI

The repository supports a local numbering workflow for maintainers who manage discussions directly in git history. The `tools/zds.zig` command, wired into the build as `zds-list`, `zds-new`, and `zds-promote`, automates it:

- list current ZDS entries and placeholder drafts
- assign the next permanent four-digit ZDS number to a placeholder draft
- rename the file from `XXXXX-<slug>.typ` to `NNNN-<slug>.typ`
- rewrite the `zds-number` metadata and transition the document into discussion
- append the metadata entry to `docs/zds/registry.typ` and the export blocks to `docs/zds/bundle.typ`

The tool performs ordinary file edits reviewed in the same change as the discussion — there is no hidden state, and every step can still be done by hand. The generated registry summary and area come from the draft's `zds-discussion` and first `zds-labels` entry; review both before committing.

That local flow is sufficient for small teams and direct-maintainer repositories. A future CI workflow can still:

- detect placeholder ZDS files that are ready for discussion
- reserve the next permanent ZDS number
- rename the file from `XXXXX-<slug>.typ` to `NNNN-<slug>.typ`
- rewrite the `zds-number` metadata

Both flows preserve a stable numbered sequence in shared history. Teams can choose local CLI assignment, CI assignment, or a combination where CI validates numbering but maintainers assign numbers intentionally.

Numbering State Diagram

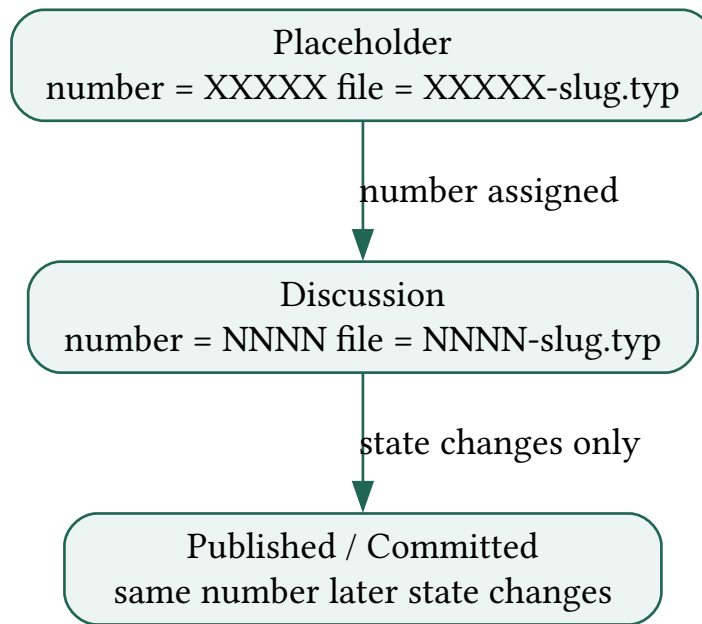


Fig. 11 The two tool flows against the same lifecycle: creating a placeholder draft, promoting it to discussion where the number is assigned, and thereafter publishing and committing, where only the state changes and the number never does.

Security Considerations

An ambiguous discussion process creates implementation drift and undocumented design assumptions. For a converter this is not cosmetic. `zenfmt` parses untrusted input by construction: its whole purpose is to accept a file someone else produced. Decisions about archive expansion limits, entry-count and decompression-ratio caps, XML entity expansion, and path traversal inside container formats are security decisions, and they must stay traceable to an explicit record rather than living as an unexplained constant in a parser.

Requiring explicit sections for scope, considerations, and alternatives reduces the chance that such limits are relaxed later by someone who cannot see why they were chosen.

References

- IETF RFCs for explicit memo structure, status, and considerations
- Oxide RFDs for engineering discussion records in a source repository
- Typst bundle export for the multi-file ZDS website and per-ZDS PDFs
- ZDS 0002, the `zenfmt` architecture and implementation record

Historical registry

These entries describe the `zenfmt` archive. Their paths are historical data. They are not local Guidedog documents.

Field	Value
number	0001
slug	zds-process
title	The Zen Discussion Process
state	committed
area	documentation
category	Process Memo
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Defines the ZDS lifecycle, numbering workflow, Typst project layout, registry, and bundle export for the zenfmt monorepo.
source	docs/zds/records/0001-zds-process.typ
html	zds/0001-zds-process.html
pdf	pdf/zds-0001-zds-process.pdf
number	0003
slug	docx-reader
title	The DOCX Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the DOCX reader
source	docs/zds/records/0003-docx-reader.typ
html	zds/0003-docx-reader.html
pdf	pdf/zds-0003-docx-reader.pdf
number	0004
slug	rtf-reader
title	The RTF Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the RTF reader
source	docs/zds/records/0004-rtf-reader.typ
html	zds/0004-rtf-reader.html

Field	Value
pdf	pdf/zds-0004-rtf-reader.pdf
number	0005
slug	xlsx-reader
title	The XLSX Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the XLSX reader
source	docs/zds/records/0005-xlsx-reader.typ
html	zds/0005-xlsx-reader.html
pdf	pdf/zds-0005-xlsx-reader.pdf
number	0006
slug	odt-reader
title	The ODT Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the ODT reader
source	docs/zds/records/0006-odt-reader.typ
html	zds/0006-odt-reader.html
pdf	pdf/zds-0006-odt-reader.pdf
number	0007
slug	pptx-reader
title	The PPTX Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the PPTX reader
source	docs/zds/records/0007-pptx-reader.typ
html	zds/0007-pptx-reader.html
pdf	pdf/zds-0007-pptx-reader.pdf

Field	Value
number	0008
slug	ods-reader
title	The ODS Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the ODS reader
source	docs/zds/records/0008-ods-reader.typ
html	zds/0008-ods-reader.html
pdf	pdf/zds-0008-ods-reader.pdf
number	0009
slug	odp-reader
title	The ODP Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the ODP reader
source	docs/zds/records/0009-odp-reader.typ
html	zds/0009-odp-reader.html
pdf	pdf/zds-0009-odp-reader.pdf
number	0010
slug	epub-reader
title	The EPUB Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the EPUB reader
source	docs/zds/records/0010-epub-reader.typ
html	zds/0010-epub-reader.html
pdf	pdf/zds-0010-epub-reader.pdf
number	0011

Field	Value
slug	pdf-reader
title	The PDF Reader
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-06
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the native PDF reader
source	docs/zds/records/0011-pdf-reader.typ
html	zds/0011-pdf-reader.html
pdf	pdf/zds-0011-pdf-reader.pdf
number	0012
slug	legacy-office-readers
title	The Legacy Binary Office Readers
state	committed
area	formats
category	Format Record
status	Committed
created	2026-08-07
updated	2026-08-08
summary	Mapping, omissions, and round-trip expectations for the DOC, XLS, PPT, and XLSB readers
source	docs/zds/records/0012-legacy-office-readers.typ
html	zds/0012-legacy-office-readers.html
pdf	pdf/zds-0012-legacy-office-readers.pdf
number	0013
slug	layered-document-ir
title	Layered Document IR and Writer Lowering
state	committed
area	architecture
category	Architectural Specification
status	Committed
created	2026-08-07
updated	2026-08-07
summary	The implemented IR v2: a layered semantic IR with sparse facets and provably deterministic writer lowering, superseding the AST and writer sections of ZDS 0002

Field	Value
source	docs/zds/records/0013-layered-document-ir.typ
html	zds/0013-layered-document-ir.html
pdf	pdf/zds-0013-layered-document-ir.pdf
number	0014
slug	python-library
title	The zenfmt Python Library: API and Implementation
state	committed
area	python
category	Implementation Specification
status	Committed
created	2026-08-08
updated	2026-08-08
summary	The implemented Python API, native boundary, uv workflow, packaging, benchmark, and release contract
source	docs/zds/records/0014-python-library.typ
html	zds/0014-python-library.html
pdf	pdf/zds-0014-python-library.pdf
number	0015
slug	wasm-and-project-site
title	Browser WebAssembly and the zenfmt Project Site
state	committed
area	web
category	Implementation Specification
status	Committed
created	2026-08-08
updated	2026-08-09
summary	The implementation blueprint for the 0.2.0 Zig WebAssembly browser release, local conversion playground, GitHub Pages project site, HTML/PDF book and ZDS help system, and reproducible benchmark dashboard
source	docs/zds/records/0015-wasm-and-project-site.typ
html	zds/0015-wasm-and-project-site.html
pdf	pdf/zds-0015-wasm-and-project-site.pdf
number	0016
slug	server
title	The zenfmt Server: REST, Streaming, and the Administered Service

Field	Value
state	committed
area	server
category	Implementation Specification
status	Committed
created	2026-08-10
updated	2026-08-10
summary	The zenfmt server: reusable CLI and service libraries, REST and streaming conversion, two-role administration on embedded zaxonlite, observability, a themed Zig WebAssembly interface, and reproducible Tika and Docling comparisons
source	docs/zds/records/0016-server.typ
html	zds/0016-server.html
pdf	pdf/zds-0016-server.pdf

Historical export manifest

The old bundle exported an HTML index, individual discussion pages, and PDFs. Its targets belonged to the zenfmt repository. It depended on that repository's Typst presentation helpers. Guidedog does not import those helpers.

This RST page records the purpose of the old entry point. The original source remains in Git history. The historical registry preserves its target paths.

See [Historical registry](#) for that data and [The Zen Discussion Process](#) for the process.