
Guidedoc: Odin conversion engine

Design, rationale, and review

01 / SOURCE

Write with meaning.

02 / STRUCTURE

Make relations clear.

03 / PUBLICATION

Read on screen and paper.

Contents

Abstract	1
Motivation	1
Purpose and boundaries	1
Memory is part of the API	4
Representation follows the work	9
Package structure and implementation order	16
Security considerations	22
Backwards compatibility	22
Alternatives considered	23
Reference implementation and open questions	23
Review history	24
Sources and design provenance	25

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

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 Guidedog 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

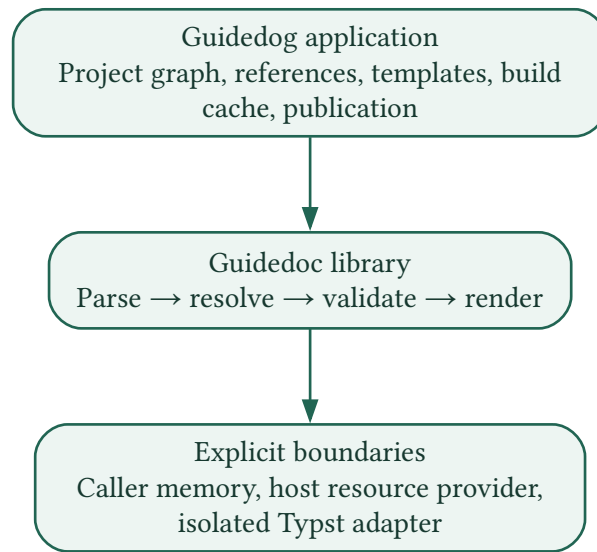


Fig. 6 Responsibility flows downward. Core conversion does not import the application.

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.

Inherited idea	Odin decision
Hot columns, cold side tables	Kind and subtree end are scanned; payloads are cold.
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 `0k` 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

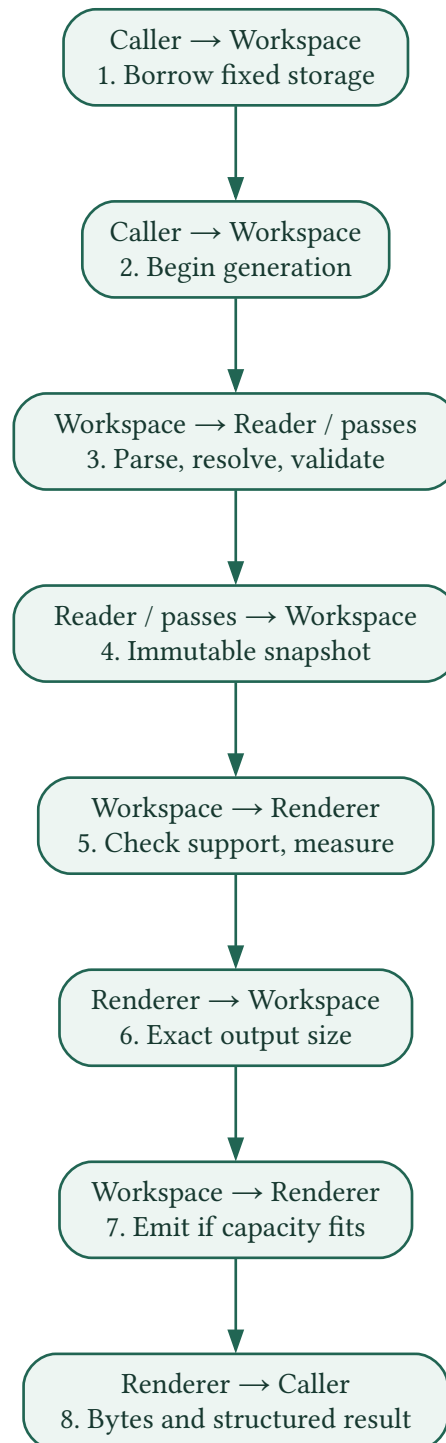


Fig. 7 No participant requests storage from an allocator during conversion.

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:intrinsics`, `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

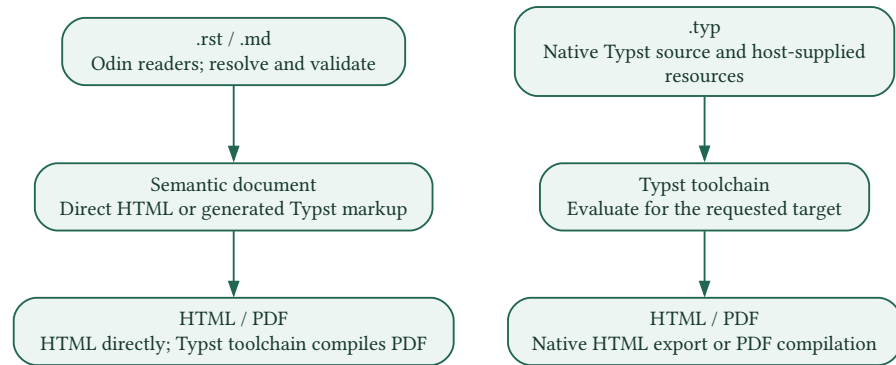


Fig. 8 Every PDF route uses Typst. Only RST and CommonMark require the Odin tree.

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>	<code>main.odin</code> ; composition and process exit only
<code>lib/core/</code>	package <code>guidedoc</code> ; memory, AST, builder,
<code>validation</code> , <code>reports</code>	
<code>lib/readers/commonmark/</code>	<code>CommonMark</code> 0.31.2; <code>myst-parser</code> 0.16.1 syntax
through a hook	
<code>lib/readers/rst/</code>	RST blocks, inline markup, directives, roles,
<code>resolution</code>	

lib/readers/sphinx/ and Markdown	Sphinx directives, roles, domains, for RST
lib/render/html/	semantic HTML encoding and escaping
lib/render/typst/	Typst source encoding, math adapter, source maps
lib/render/text/	plain text, as Sphinx's text builder writes it
lib/defaults/	explicit built-in reader and renderer registry
lib/host/ publication	storage sizing, retries, resource provider,
lib/typst/ the bridge)	adapter to the Typst backend (installed CLI or
lib/cli/	options, commands, terminal diagnostics
lib/gds/	metadata, lifecycle, registry, promotion, recovery
lib/project/ builders, themes	discovery, configuration, symbols, cache,
lib/jinja/	Jinja templates for pages; independent of Guidedoc
lib/highlight/ independent	code highlighting with Pygments' classes;
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/ Typst compiler	narrowly scoped Rust-to-C ABI bridge to the
native/graphviz/ lib/graphdog	script that builds Graphviz statically for
tests/	conformance, malformed input, goldens, integration
tools/	source-limit checks and documentation commands
docs/gds/ and archive	standalone discussions, index, template,

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 `guidedog` 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.

Concern	Evidence required
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.
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 resolu-

Concern	Evidence
	tion; 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.