Guidedog Discussions GDS 0003

Guidedoc: the conversion engine

STATE
committed
TYPE
Standards Track
AUTHORS
Vikrant Rathore
CREATED
2026-09-29
UPDATED
2026-10-31
DISCUSSION
Local review; no external discussion URL assigned
LABELS
architecture

Record status

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

Contents

1. Abstract 1
2. Motivation 1
3. Purpose and boundaries 2
4. Memory is part of the API 3
5. Representation follows the work 7
6. Crate structure and implementation order 11
7. Security considerations 14
8. Backwards compatibility 14
9. Alternatives considered 14
10. Reference implementation and open questions 15
11. Review history 15
12. Sources and design provenance 16

1. Abstract

This proposal defines Guidedoc, a reusable Rust library with a core that allocates nothing in steady state, a flat AST in typed tables, 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.

2. Motivation

The existing zenfmt design supplies useful document and ownership ideas, but its Zig APIs and allocator strategy do not express the intended library contract. docodin, an 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 engine from resource acquisition and the Typst runtime.

3. 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.

3.1. Three responsibilities

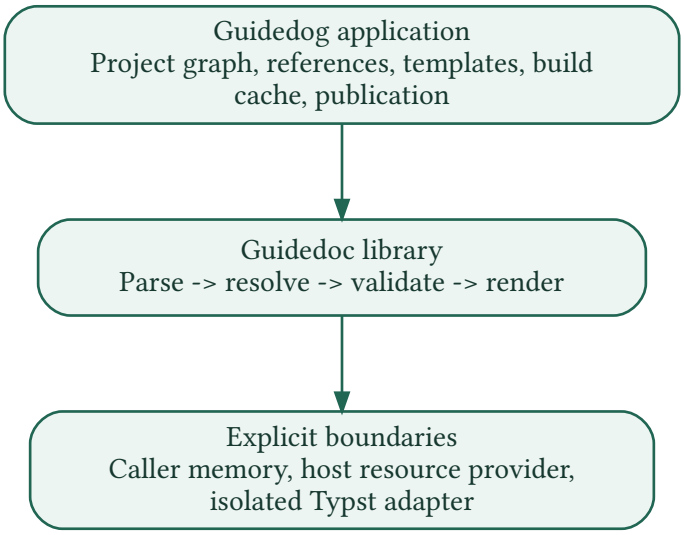

Fig. 6 Responsibility flows downward. Core conversion does not import the application.

The 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. A conversion keeps its tables in a reused workspace, so a repeated conversion allocates nothing. The Typst compiler has its own memory management, and that guarantee does not extend into it.

3.2. 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 experience. These are design inputs, not reused code.

Inherited idea	Guidedoc decision
Flat semantic tree (zenfmt)	Typed IDs and flat tables (`Vec`) indexed by them.
Handles, not pointers (docodin)	`NodeId(u32)` newtype IDs; bounds checks stay on.
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.

Inherited idea	Guidedoc decision
Elm-style reports	Structured evidence and concrete next actions.
Compile-time bundles	Ordinary descriptor constants in an immutable registry.
Arena growth	Workspace tables reused between conversions; growth bounded by `Limits`.
Mandatory adjacent manifest	Opt-in host preservation artifact.

We do not translate Zig's comptime type factories or generated routing matrix. Crate dependencies and function values are enough to connect a reader and a writer. A small indirect call per stage is acceptable; dispatch per character is not.

3.3. Terminology

A **workspace** owns the reusable storage of one active conversion. A **document** is a validated immutable view of a workspace snapshot, borrowed from the workspace. 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.

3.4. Non-negotiable contracts

- A conversion in a reused workspace allocates nothing: parsing, resolution, validation, rewriting, render checking, diagnostic construction, and writing reuse the capacity the workspace kept. Growth that input drives is bounded by `Limits` and reserved with `try_reserve`, so running out is a result, not an abort.
- 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 explicit stacks held by the workspace. 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.

4. Memory is part of the API

"No allocation in steady state" means that a conversion reuses storage the workspace already holds and asks the allocator for nothing more. It does not mean zero memory, zero copying, or an unbounded document in a fixed buffer. The first conversion in a new workspace grows its tables. Later conversions clear them with `clear()`, which keeps their capacity, and fill them again.

The host chooses how many workspaces to keep: one per thread for a build, one per request for a server, a fresh one for a single command. A workspace is never shared by concurrent conversions, and the registry it converts with is an immutable value.

4.1. Storage regions and lifetime

Region	Contents	Valid until
Input	The main source (copied into a reused buffer) and included sources (`Arc<str>`)	The next conversion
Snapshot A / B	Node table, payload tables, attributes, text pool	The next rebuild of that snapshot
Scratch	Adapters' reusable working structs, a pool per type	Taken and given back within a stage
Reports	Diagnostic records, bounded by `Limits::max_reports`	The next conversion
Output	The rendered bytes, borrowed from the workspace	The next conversion

The lifetimes in the third column are checked by the compiler. A `Document<'w>` and the output of `convert_text` borrow the workspace, so a stale or foreign handle does not compile.

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 snapshot, so validation checks one pool and a rebuild copies nothing it cannot own. Validation also checks that the pool is UTF-8 and that every `Text` lies on character boundaries.

Scratch is a store of reusable working structs. An adapter takes a struct of its own out of the workspace (`ctx.take_scratch::<T>()`), clears what it uses, and gives it back (`ctx.put_scratch(value)`). Its `Vec`s and `String`s keep their capacity between conversions, so a repeated conversion allocates nothing. `NameMap` is a name table that keeps its key storage when cleared.

Two snapshots permit rebuild passes. A pass reads A and writes B, validates B, then makes B current. On failure, A remains valid. An unchanged pass returns A immediately. There is no persistent sharing or copy-on-write mechanism in version one.

4.2. Small vocabulary, ordinary Rust

Types are `CamelCase` and functions `verb_noun`. We borrow the clarity of raylib's small verbs: plain-data structs with `Default` filled with struct-update syntax, a one-call path beside the staged one, and no hidden global engine state. There are no fluent builders, class hierarchies, required macros, or allocator parameters.

```
use guidedoc::{convert_text, Workspace};

fn main() {
    let registry = guidedoc_defaults::registry();  // the built-in readers and renderers
    let mut ws = Workspace::new();                 // reuse it: buffers are kept
    match convert_text(&mut ws, &registry, "commonmark", "html", "# Welcome\n") {
        Ok(html) => println!("{html}"),            // borrows ws until the next convert
        Err(fail) => eprintln!("{fail}"),          // title, message, and hint
    }
}
```

A workspace owns its storage and frees it when dropped. No close or destroy call is needed. Every fallible call returns `Result<_, Fail>`. Callers needing inspection use the staged interface.

```
pub enum Status { Ok, InvalidInput, Unsupported, Policy, Capacity, Limit, Cancelled, Internal }
pub enum Region { None, Nodes, Text, Attributes, Lists, Links, Cells, Sources,
                  Scratch, Reports, Output, SourceMap }
pub struct CapacityNeed { pub region: Region, pub minimum: u64, pub exact: bool }
pub struct Fail { pub status: Status, pub need: CapacityNeed, pub diagnostic: Box<Diagnostic> }

pub fn convert_text<'w>(ws: &'w mut Workspace, registry: &Registry, reader: &str,
                        renderer: &str, text: &str) -> Result<&'w str, Fail>;
pub fn convert<'w>(ws: &'w mut Workspace, registry: &Registry, request: &Request<'_>)
    -> Result<&'w str, Fail>;
pub fn read_document<'w>(ws: &'w mut Workspace, reader: &Reader, request: &Request<'_>)
```

```
    -> Result<Document<'w>, Fail>;
pub fn transform_document<'w>(ws: &'w mut Workspace, passes: &[Pass<'_>],
                              limits: &Limits) -> Result<Document<'w>, Fail>;
pub fn check_render(ws: &mut Workspace, renderer: &Renderer, request: &Request<'_>)
    -> Result<usize, Fail>;
pub fn render_document<'w>(ws: &'w mut Workspace, renderer: &Renderer,
                           request: &Request<'_>) -> Result<&'w str, Fail>;
```

The staged functions work on the workspace's current document, so there is no handle to pass between them. Render checking and emission use the same immutable snapshot, renderer, and options. Emission repeats the check.

`Request` includes limits, policy, render options, optional passes, a resource provider, and a cancellation probe. Defaults come from explicit `default_limits` and `default_policy` functions, and `default_request(name, text, reader, renderer)` assembles a complete request from them. Zero-valued limits do not silently mean unlimited.

A reader's own settings travel in `ReadOptions::extension`, a `ReadExtension` made by `read_extension(&settings)`. It carries a reference and its type's name. A reader takes them back with `ctx.reader_settings::<T>()`, which refuses a value of another type with `InvalidInput` and a diagnostic naming both types (`core.read.settings_type`) before anything is read. So settings meant for one reader are never taken for another's. The mechanism stays generic, so readers of other registries use it with their own types. `guidedoc-defaults` adds typed helpers for the built-in readers.

4.3. 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 `ResourceProvider`, a trait with one method, `provide(from, name)`. The reader asks for a name relative to the including source. The provider returns the text 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 the workspace's 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 (`gd_host::Files`) confines reads to approved roots with symlink-aware checks, as GDS 0002 says.

4.4. 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 caller provides `Limits` (`max_nodes`, `max_text_bytes`, `max_output_bytes`, `max_reports`, `max_depth`, `max_work`, `max_include_depth`, `max_included_bytes`, `max_expansion`, `stack_bytes`). Reaching one is a `Limit` or `Capacity` result whose `need` names the region and the minimum needed at that point. This is a lower bound unless `exact` is true. A refused `try_reserve` is a `Capacity` result too. Arithmetic overflow is checked before addition, multiplication, or slicing.

A renderer has one encoding function and two modes: **check** reports unsupported content without output, and **write** stores the bytes. Writing runs once into an emitter with a byte limit that keeps counting past it, so an oversized output still reports its exact size. Parsing or render checking failure leaves no output. A successful write checks that the output is UTF-8 before returning it.

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

4.5. Conversion sequence

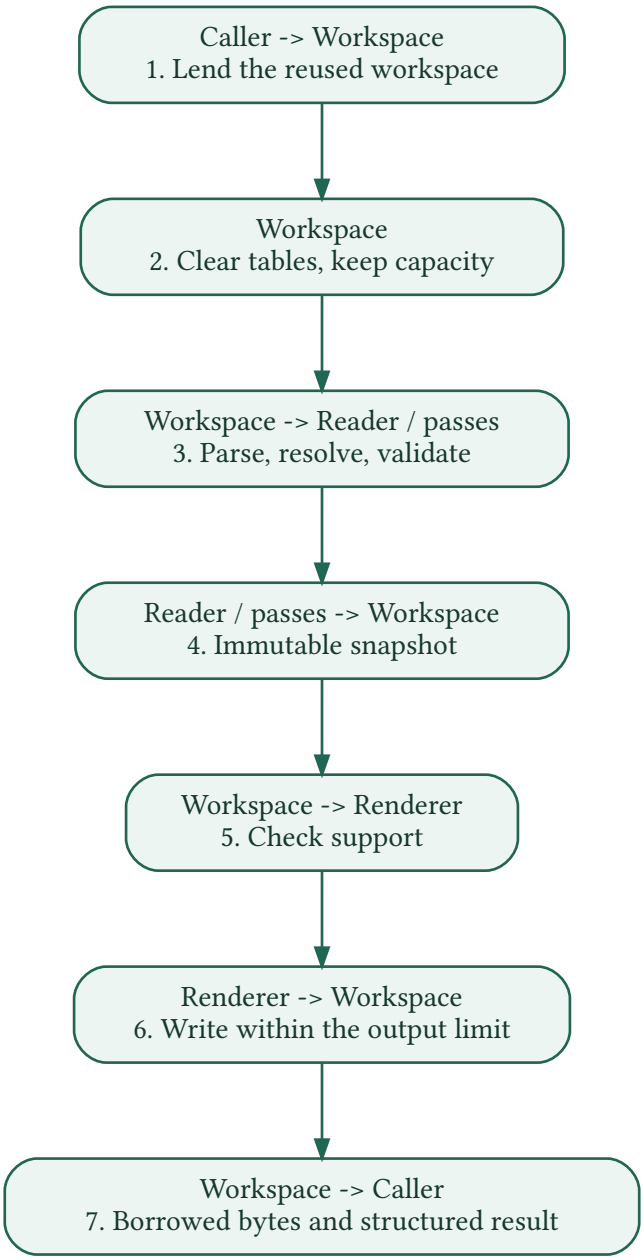

Fig. 7 In steady state no participant asks the allocator for storage during conversion.

Failure at steps 3 through 6 returns a diagnostic and prevents output. The next conversion releases all borrowed views together. Concurrent callers use separate workspaces.

4.6. Proving the allocation boundary

Rust has no per-call allocator to take away, so a counting global allocator observes conversions. `steady_state_conversion_does_not_allocate` (`crates/guidedoc/tests/services.rs`) converts the same text twice in one workspace and requires the second conversion to allocate nothing. `tests/integration/tests/allocations.rs` does the same with every built-in reader (RST, Sphinx, MyST, strict CommonMark) into every renderer. It requires that a repeated conversion allocates no more than the time before, and no more than a ceiling recorded for the pair. The ceilings are a ratchet: a change lowers one when it removes allocations and never raises one without a reason. The few allocations that remain are explained in `docs/manual/internals/memory.rst`.

The engine crates (`guidedoc*`) use only the standard library. Cargo enforces their dependency direction, because a crate can use only what its `Cargo.toml` declares.
Helpers that a host also wants in an allocating form keep a form that writes into a caller's buffer. `normalize_nfd_into` writes Unicode NFD into a caller's `String`. The project host uses it for its index keys.

5. 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 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 core.

5.1. The conversion routes

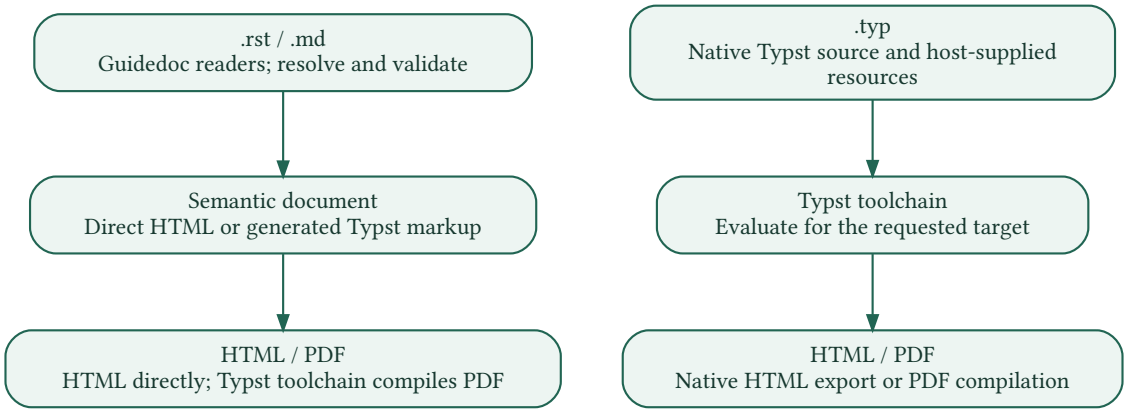

Fig. 8 Every PDF route uses Typst. Only RST and CommonMark require the document tree.

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.

5.2. Flat structure, meaningful types

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

```
pub struct NodeId(pub u32);
pub struct Text { pub start: u32, pub len: u32 }
pub struct SourceSpan { pub source: SourceId, pub start: u32, pub end: u32 }
pub struct Node {
    pub kind: NodeKind,          // hot: scanned by traversal
    pub flags: NodeFlags,
    pub subtree_end: u32,        // hot: exclusive preorder end
    pub value: i32,              // small per-kind scalar: heading level, number
    pub text: Text,              // leaf content, URL, label, or name
    pub payload: u32,            // row in the kind's payload table, or NONE
    pub span: SourceSpan,
    pub attributes: AttributeRange,
}
// Snapshot::nodes is a Vec<Node> in preorder; nodes[0] is the Document.
```

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
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 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 their 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.

5.3. 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.

Symbol tables are `HashMaps` with collision-resistant hashing and a capacity bound. Growth past the bound is a capacity error, and a duplicate name is a `SymbolClash`. So hostile input cannot cause unbounded work.

5.4. 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 a second snapshot nor a pass. Substitution cycles, duplicate targets, and unresolved references have dedicated reports with both source sites where two exist.

5.5. 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.

5.6. 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 `match` over `NodeKind`, with no `_ =>` arm. 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. An integration test renders a document holding every kind with every built-in renderer (`crates/guidedoc-defaults/tests/kinds.rs`). Payload tables, 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.

5.7. 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 + 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 snapshots and is linear in copied nodes and text. Version one favors a few clear stages over persistent trees, SIMD, or speculative parallel parsing.

5.8. Recursion is bounded by the depth limit and the stack budget

Decision. The builder's open nodes, validation, copying between snapshots, anchors, the HTML renderer's walk, and the CommonMark reader keep nesting in explicit stacks held by the workspace. The RST reader (a body inside a directive, list, or quote parses its own body), the C++ declaration parser of `guidedoc-cpp` (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 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 `StackGuard` (`crates/guidedoc/src/stack.rs`, made by `stack_guard`) records the address of a local where the reader or renderer starts, and `stack_exhausted` compares the address of a local in the current frame with it. The measure is exact whatever the functions' 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). A build without optimization has larger frames 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 C++ parser also bounds its steps by the length of the declaration (`STEPS_PER_BYTE` 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.
`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. `gd-cli` replaces those hints with the flags that set them when it shows the reports.
The host runs conversions on threads whose stack it knows. `gd_host::run_workers` and `run_with_stack` start threads with `WORKER_STACK_BYTES` (8 MiB). `stack_budget_limit` leaves 2 MiB of such a stack (and at least half of it) for the frames around the guarded recursion, so the largest budget is 6 MiB. 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 (`default_stack_budget`).
Alternatives. Rewriting the recursive readers and renderers around explicit stacks would remove the machine-stack dependence entirely. But it touches most of their functions 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. A host that wants more nesting raises `stack_bytes` and runs the conversion on a larger thread itself.

5.8.1. The supporting libraries

The libraries Guidedog is built from do not depend on `guidedoc` (their `Cargo.toml` files do not declare it), so each bounds its own recursion. Each uses 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.

- `gd-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` is a frame-address guard started by the first public call on the environment and checked by the parser's nesting and every evaluation 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. The guard is only the backstop for hostile ones. Walks over values (repr, equality, ordering, `tojson`) stop at 200 levels (`MAX_VALUE_DEPTH`). repr prints `[...]` for a list that contains itself, as Python does.

- `gd-pyscan`: tree-sitter parses with a heap stack, so any nesting parses. `deeper_than` measures a tree with a cursor, without recursion. 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 is 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.
- `gd-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.
- `gd-odindoc`: its parser of Odin source counts how deeply it recurses (nested brackets, type prefixes, casts, ternaries, `when` blocks) and gives a file up past `MAX_NESTING` (160), with the warning `odin.autodoc.nesting`. The `when` conditions it evaluates are followed `MAX_WHEN_DEPTH` (64) levels deep.
- `gd-rustdoc`: tree-sitter parses Rust with a heap stack too. Module trees, re-export chains, and items are bounded in depth and count.
- `gd-gettext` bounds its plural expressions (100 levels, 128 nodes). Costs a hostile catalog could square are linear in it (or n log n). A PO field's continuation lines are gathered in a builder and copied once. A compiled catalog is validated once at load, and its hash table lookups stop after 64 probes. The fuzzy search of `merge` walks a length-sorted, trigram-ranked candidate list that stops where no candidate could win.
- `gd-highlight` keeps nested lexer states in an explicit stack of 24 frames. `gd-graphdog` does not recurse (Graphviz's parser reports DOT nested 10,000 deep as an error).
- `gd-score`, `gd-lilypond` and `gd-engrave` (GDS 0015): rational time uses checked `i64` arithmetic; the LilyPond parser bounds nesting depth (64), source size (256 KiB), tokens, events and includes; and the engraver bounds systems, glyphs and skyline/layout scratch buffers.

Alternatives. One shared guard crate would remove the small repetition of the frame-address guard in `gd-jinja` and `gd-pyscan`, but would make every library depend on it. The guard is a dozen lines.

6. Crate structure and implementation order

Crate boundaries express dependency direction. File boundaries keep functions and explanations small enough to read in one sitting.

```
cmd/guidedog/                the single Guidedog executable (GDS 0012); thin over gd-cli
crates/guidedoc/             the engine core: AST, builder, validation, diagnostics, registry
crates/guidedoc-commonmark/  CommonMark 0.31.2 and MyST syntax
crates/guidedoc-rst/         RST blocks, directives, roles, resolution
crates/guidedoc-rst-inline/  RST inline markup
crates/guidedoc-sphinx/      Sphinx directives, roles, domains, for RST and Markdown
crates/guidedoc-cpp/         C++ declarations for the C++ domain
crates/guidedoc-html/        semantic HTML encoding and escaping
crates/guidedoc-typst/       Typst source encoding, math adapter, source maps
crates/guidedoc-text/        plain text, as Sphinx's text builder writes it
crates/guidedoc-rst-writer/  reStructuredText writer for round-trip and conversion
crates/guidedoc-defaults/    the built-in registry and the highlighter adapter
crates/gd-host/              sessions, budget, confinement, publication, workers
crates/gd-report/            Elm-style reports, text and JSON renderers
crates/gd-typst/             the Typst compiler, embedded, or an installed fallback
crates/gd-cli/               options, commands, terminal diagnostics
crates/gd-discussion/        metadata, lifecycle, registry, promotion, recovery
crates/gd-conf .. gd-project the project builder (GDS 0006)
crates/gd-jinja/             Jinja templates for pages; independent of Guidedoc
crates/gd-highlight/         code highlighting with Pygments' classes; independent
```

```
crates/gd-graphdog/          Graphviz's C library for graphs; independent
crates/gd-score .. gd-engrave native LilyPond music model, reader and engraver (GDS 0015)
crates/gd-gettext/           gettext catalogs for translations; independent
tests/integration/           conformance, malformed input, goldens, integration
tools/                       source-limit, music-corpus and repository tasks
docs/gds/                    standalone discussions, index, template, and archive
```

`guidedoc` depends on no reader, renderer, terminal, or Typst crate. Readers and renderers depend on `guidedoc` and use only the standard library. `guidedoc-defaults` depends on those adapters and assembles a registry. Host and CLI compose the library. Typst remains an explicit host dependency, inside `gd-typst` only. Embedders can depend on `guidedoc` and a selection of adapters without the CLI. Guidedog, the application, is `cmd/guidedog` with `gd-cli`, the project crates, and `gd-host`. Guidedoc, the engine, is `guidedoc` with the readers and renderers. The libraries a Guidedog build also uses (Jinja, the highlighter, graphdog for Graphviz, and gettext) depend on no Guidedog or Guidedoc crate, so other programs can use them alone. Only the small adapter in `guidedoc-defaults` joins the highlighter to the engine. Cargo enforces these rules, because a crate can use only what its `Cargo.toml` declares, and `tests/integration/tests/gates/dependencies.rs` checks the declared direction.

Every public item has a doc comment (`///`) that states its contract as GDS 0001 asks: what it borrows or copies, how long results live, and how it fails. Each crate's `README.md` is a short cheat sheet of its API, and its examples are doc tests, so they cannot fall out of date with the API. Items are private by default, and each crate exports a small surface from `lib.rs`.

The readers and renderers are narrowed the same way. Each exports its descriptor constant (`READER` or `RENDERER`), its settings types with their defaults, and, where another crate builds on it, a named interface: the RST reader's extension interface (the directive and role handlers, the parsed directive and its options, 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. Diagnostics are identified by their stable `code`.

Every other library is narrowed the same way, so each can be used on its own through the small API its README lists. Where narrowing found an unclear owner, the contract was fixed rather than described: Rust's ownership makes the owner of every returned value explicit in its type.

6.1. Host surface

`gd-host` exports what the command line, the project builder, the engine's test suites, and an embedder need to run the engine with files and a budget: sessions (`Session` with `run`, `read`, `document`, `reports`), files (`read_source`, `read_file`, `same_contents`, `each_chunk`), includes and confinement (`Files`, a `guidedoc::ResourceProvider`; `canonical_roots`, `confined`, `confinement`, `containing_root`), publication (`publish`, `Stage`, which is removed on drop unless committed, `write_unsynced`, `sync_all`, `folders_of`, `exchange`), the budget (`Budget`, `Ledger`, `limits_within`), diagnostics (`describe`, `describe_all`, `exit_code`, `display_path`, `Problem`), interrupts, threads with known stacks, the crash reporter, and the `guidedog convert` route (`convert_artifact`). `crates/gd-host/README.md` lists the surface.

One ownership rule covers the crate: what a function returns is owned by the caller, and arguments are borrowed for the call. A session owns what its results refer to until it runs again or is dropped. Failure is a `Problem` or a `Result` error, never a panic. A budget refusal is `host.budget`. The project builder's surface is recorded with the project (`0006-sphinx-replacement`, "Project surface").

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.

`cargo run -p check` enforces the source limits. Lines have at most 99 characters and never more than 108. Files have at most 1,408 lines. Functions have 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, 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.

```
cargo build --release -p guidedog-cmd
cargo test -p guidedoc
cargo test --workspace
```

Typst is 0.15.1.

6.2. Adapter contracts

A registry is an immutable pair of `Vecs`: 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. `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 reader.

A descriptor is a constant holding ID, name, version, conformance status, and a function (`Reader { id, name, version, conformance, read, read_inline }`). Options are typed: `ReadOptions` carries the reader's own settings as a typed `ReadExtension`, and `RenderOptions` selects standalone or fragment output, the reference profile, and a document title. Registry validation checks unique IDs and required metadata. `convert` runs it before reading. Adapters retain no workspace references between calls.

The engine's exported names form two surfaces. The conversion surface holds workspaces, requests and their defaults, `convert` and the staged functions, failures, diagnostics, and the read-only `Document` view. The adapter and pass surface holds descriptors, contexts, the builder, the emitter, the scratch store, symbol tables, anchors, and shared text helpers. Three services for hosts stand beside them: documents saved as bytes for a build cache, Unicode NFD, and line and column positions. Everything else is private to the crate. The API reference (`docs/api`) lists the public names.

A pass is a closure `Fn(Document, &mut Builder) -> PassOutcome`. It receives an immutable source snapshot and a destination builder. It returns `Unchanged`, `Rebuilt`, or `Failed`. The engine runs passes in order and validates each rebuilt snapshot. The builder exposes `begin_node`, `end_node`, `with_node`, `add_leaf`, `add_text`, `add_attribute`, and typed payload functions. Every function returns a `Result`. A failed builder is poisoned: every later call returns the same failure, so `?` propagates it and ignoring one capacity failure cannot publish a truncated document. Unbalanced nodes fail finalization.

`gd_host::convert_artifact` takes an explicit source, target, resource provider, budget, staging destination, and optional Typst backend. Resource acquisition and Typst compilation may allocate. A caller requesting PDF without a backend gets an actionable result.

6.3. Diagnostics as static records

```
pub struct Diagnostic {
    pub code: &'static str,     // stable: "rst.reference.unresolved"
    pub severity: Severity,
    pub status: Status,
    pub title: &'static str,
    pub message: &'static str,  // template: "No target is named {0}."
    pub hint: &'static str,     // how to fix it
    pub span: SourceSpan,
    pub args: [Arg; 2],         // integer, source span, or static text
}
```

Messages are static templates with at most two typed arguments, so constructing a report never formats text. `Diagnostic::format_into` fills a template into a caller's buffer when the host shows it. A source-span argument is quoted by the host renderer from the source table. When report storage fills (`Limits::max_reports`), `Diagnostics::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 primary record.

6.4. Correctness before tuning

Flat tables, 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 memory, table sizes, 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.

6.5. Acceptance matrix

Concern	Evidence required
Memory boundary	A repeated conversion in one workspace allocates nothing in the core and no more than its recorded ceiling with every reader and renderer; the engine crates use only the standard library.
AST safety	Malformed IDs, ranges, nesting, and table spans fail validation cleanly; stale handles do not compile; 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.
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 limits 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.

7. Security considerations

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

8. Backwards compatibility

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. Saved documents (the build cache's objects) carry a format version, now 2, and a cache of another version is rebuilt rather than misread. Changes to a published node schema require a separate GDS and an explicit compatibility decision.

9. Alternatives considered

A pointer tree (`Rc<RefCell<..>>` nodes) complicates ownership and validation. Flat tables retain 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.
Caller-sized fixed storage, with a capacity result and a retry, was considered for the workspace. It makes every caller size regions it cannot predict, while reused `Vec`s reach the same steady state with no sizing at all. `Limits` keep the bound that fixed storage gave.
Reusing docodin's RST engine was considered. It is complete and measured, but it reproduces Docutils' document model and output byte for byte. Guidedoc needs its own semantic tree, 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 rebuilds and validation harder than the linear copy it saved. A custom Typst evaluator would duplicate the toolchain and is not proposed.

10. Reference implementation and open questions

The crates listed above implement this proposal. `crates/guidedoc/README.md` is the embedder's cheat sheet and `docs/manual/internals/ast.rst` the normative node mapping. Beside the staged interface, `convert_text` makes the common case one workspace and one call.
Evidence:

Concern	Evidence
Memory boundary	`steady_state_conversion_does_not_allocate`; `tests/integration/tests/allocations.rs` with a ceiling for every reader and renderer pair.
AST safety	Malformed ends, nesting, payloads, and illegal parents fail validation; builders poison on the first failure.
CommonMark	All 652 examples of 0.31.2 in the reference profile; hostile patterns end as results within the work limit (`docs/manual/internals/commonmark-conformance.rst`).
RST	Tests over blocks, directives, inline markup, roles, and resolution; coverage ledgers (`docs/manual/internals/rst-conformance.rst` and its ledgers); 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.
Math	LaTeX translations compile; unknown commands are refused by name.

The RST ledgers 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. `ListInfo::delimiter` is one byte, so the bullets •, ‣ and ⁃ travel in a `bullet` attribute.

11. Review history

29 September 2026: initial proposal. Review removed writer lowering, selected full RST and CommonMark, selected a flat AST in typed tables, 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 taken from docodin; make node text owned by the snapshot pool; replace up-front resource entries with a provider; resolve references in side tables so a second snapshot is optional; require readers to emit in preorder from skeletons; define one encoder with check and write modes; add the LaTeX-to-Typst math adapter; replace the options schema with typed options; and specify static diagnostic records.
29 September 2026, implementation: record the implemented crates and evidence, the one-call entry point, the `target-name` attribute for embedded aliases, and project links and anchor prefixes for multi-document books.

29 September 2026, engine review: prove the allocation boundary with every reader and renderer; report content that does not fit instead of cutting it; validate registries in `convert`; make engine internals private and name the two public surfaces; add `default_request` and typed read options; 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 matches. 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), and add `convert --stack-kib`. Later again: narrow every library to the API its cheat sheet documents.
30 September 2026, API and host reviews: narrow the readers, the renderers, and the host as the engine was narrowed, and give every public item a contract.

12. 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 copy.
- Neighboring `../docodin/docs/dod/0004-data-oriented-storage.rst`, `0012-parser-research-0-2-0.rst`, and `0013-source-spans.rst`: handles, struct-of-arrays storage, and span rules.
- The Rust Reference: ownership, borrowing, and lifetimes.
- 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: compiler stages and resource environment.

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. Library and integration suites;
embedded-backend validation. Deferred features are not claimed as implemented.
```