
Sphinx-style projects and recoverable publication

Design, rationale, and review

01 / SOURCE

Write with meaning.

02 / STRUCTURE

Make relations clear.

03 / PUBLICATION

Read on screen and paper.

Sphinx-style projects and recoverable publication

Version 0.2.0

© 2026, Vikrant Rathore

Oct 11, 2026

Contents

Abstract	1
Motivation and scope	1
Specification	2
Rationale and alternatives	14
Security and operational considerations	14
Backwards compatibility	15
Acceptance criteria and implementation status	15
Open questions	16
Review history	16
References	16

Number: 0006
Title: Sphinx-style projects and recoverable publication
State: committed
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-29
Updated: 2026-10-11
Discussion: No external discussion URL assigned
Labels: architecture, cli, build

Abstract

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

Motivation and scope

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

Principles

- **Capability, not imitation.** Supported authoring features follow Odin’s idioms. Arbitrary Python configuration and extensions remain outside the support boundary.

- **Data over code.** Settings and object declarations are checked data. Python configuration is never executed. Trusted PDF templates run in Typst; `--untrusted` substitutes Guidedog's template and omits raw Typst.
- **Stable where users depend on it.** Page paths, anchors, `objects.inv`, and numbering match Sphinx, so links into and out of a project keep working.
- **Better where Sphinx is weak.** A failed build publishes nothing; diagnostics explain and are the same on every build; files outside the project are read only when allowed; memory is bounded.
- **Migrate, don't emulate.** Guidedog is a new system that supports Sphinx projects, with caveats. Sphinx fidelity is pursued so existing projects build, but it is not the final goal. Where Odin's idioms conflict with Sphinx's Python runtime (executed `conf.py`, extensions' `setup()`, event hooks, imported code), Guidedog does not emulate the runtime: it has a migration path instead. `guidedog migrate` converts literal settings and declared object types today; later, a Guidedoc migration plugin will help transfer supported Sphinx authoring features, and a project that cannot be migrated is told exactly what needs converting by hand.
- **Honest.** Every difference from Sphinx is documented, with its reason, in the manual's support matrix; a comparison harness against pinned Sphinx measures the rest.

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

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

Specification

Compatibility target

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

mean copying a runtime quirk or a weaker design, Guidedog does the better thing and the support matrix records the difference and why.

Command line

```
guidedog quickstart [DIR] [-q] [-p PROJECT] [-a AUTHOR] [-v VERSION] [-r RELEASE]
                                [-l LANGUAGE] [--sep] [--suffix .rst] [--no-batch]
guidedog build [BUILDER] [PROJECT] [options]          # make-mode: _build/BUILDER
guidedog build -b BUILDER SOURCEDIR OUTPUTDIR [FILES] [options] # sphinx-
build form
guidedog clean [PROJECT]
guidedog serve [PROJECT] [--port 8000]                # autobuild: rebuild on change
guidedog migrate [conf.py]                             # literal settings
to conf.toml
```

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

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

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

Project layout

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

```
docs/                                docs/
  conf.toml                          source/
  index.rst                          conf.toml
  .gitignore                         index.rst
  _static/                           _static/
```

```

guidedog.css      guidedog.css
guidedog.js      guidedog.js
_templates/      _templates/
  layout.html    layout.html
  book.typ       book.typ
_build/          .gitignore
                 build/

```

`index.rst` is Sphinx's blank root document: a comment, a title, one paragraph pointing to the reStructuredText documentation, and an empty root `toctree` with `:maxdepth: 2` and `:caption: Contents:`. Prompts ask what `sphinx-quickstart` asks; `-q` takes defaults and flags. `.gitignore` ignores the build directory and Guidedog's caches.

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

conf.toml

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

```

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

```

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

- **Project:** `project`, `author`, `copyright`, `version`, `release`, `today`, `today_fmt`.
- **General:** `root_doc` (`master_doc`), `source_suffix`, `source_encoding` (UTF-8 only), `exclude_patterns`, `include_patterns`, `include_roots`, `templates_path`, `extensions`, `rst_prolog`, `rst_epilog`, `default_role`, `primary_domain`, `highlight_language`, `pygments_style`, `numfig`, `numfig_format`, `numfig_secnum_depth`, `math_number_all`, `math_eqref_format`, `math_numfig`, `nitpicky`, `nitpick_ignore`, `nitpick_ignore_regex`, `show_authors`, `add_function_parentheses`, `add_module_names`, `toc_object_entries`, `trim_footnote_reference_space`, `smartquotes`, `smartquotes_action`, `smartquotes_excludes`, `keep_warnings`, `suppress_warnings`, `needs_sphinx` (ignored).
- **Language:** `language`, `locale_dirs`, `today_fmt` is localized.
- **HTML:** `html_theme`, `html_theme_options`, `html_title`, `html_short_title`, `html_logo`, `html_favicon`, `html_static_path`, `html_extra_path`,

`html_css_files`, `html_js_files`, `html_last_updated_fmt`, `html_permaLinks`, `html_permaLinks_icon`, `html_use_index`, `html_split_index`, `html_copy_source`, `html_show_sourcelink`, `html_sourcelink_suffix`, `html_show_copyright`, `html_show_sphinx` (shows Guidedog), `html_baseurl`, `html_file_suffix`, `html_link_suffix`, `html_secnumber_suffix`, `html_codeblock_linenoStyle`, `html_context`, `html_additional_pages`.

- **PDF** (the LaTeX settings' counterpart, typeset by Typst): `pdf_documents`, `pdf_paper_size`, `pdf_logo`, `pdf_toplevel_sectioning`, `pdf_show_urls`, `pdf_preamble` (a Typst file for package imports and show rules), `pdf_packages` (download or offline), `pdf_fonts` (system or embedded), `pdf_font_paths`.
- **Built-in extensions:** `extensions` accepts `sphinx.ext.mathjax`, `sphinx.ext.imgmath`, `sphinx.ext.todo`, `sphinx.ext.githubpages`, `sphinx.ext.ifconfig` (with literal expressions), `sphinx.ext.extlinks` (with its dictionary), `sphinx.ext.autosectionlabel` (with `autosectionlabel_prefix_document` and `autosectionlabel_maxdepth`), `sphinx.ext.graphviz` (with `graphviz_output_format` and `graphviz_dot_args`), `sphinx.ext.intersphinx` (see below), and `sphinx.ext.autodoc` (with Sphinx's `autodoc_*` settings, `autoclass_content`, and `autodoc_source_paths`, the folders its Python source is read from), which Guidedog implements natively. Any other extension is an error that names it and says it needs the future plugin system.

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

Builders

Builder	Output
<code>html</code>	one page per document; <code>genindex.html</code> , <code>search.html</code> , <code>searchindex.js</code> , <code>py-modindex.html</code> when modules exist, <code>objects.inv</code> , <code>_static/</code> , <code>_sources/</code> , <code>_images/</code> , <code>_downloads/</code>
<code>dirhtml</code>	as <code>html</code> , with <code>docname/index.html</code> pages
<code>singlehtml</code>	the whole toctree on the root document's page; each document's anchors carry its key, so links, the index, and search reach the right one
<code>pdf</code>	one Typst-typeset PDF per <code>pdf_documents</code> entry, in <code>_build/pdf</code>
<code>text</code>	plain text per document, in <code>_build/text</code>

Builder	Output
<code>gettext</code>	message templates (<code>.pot</code>) per document or per folder
<code>dummy</code>	read and link every document, reporting what linking reports; no output

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

The build environment and incremental builds

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

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

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

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

Publication

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

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

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

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

- While the build runs, every file it changes is written to the next folder (through a temporary file and a rename); a file whose bytes are already published is only recorded. Nothing in the live output changes.
- When the build succeeded, the manifest and the environment are written beside their destinations (`<builder>.outputs.json.next`, `<builder>.environment.json.next`, each only when its bytes differ), and the record each will replace is kept as a hard link (`.previous`), to put back if the commit is undone.
- The next folder is completed into the whole next output (`complete_next`): every file of the live output that is not obsolete (made by the previous generation and not by this one) is hard-linked into it, or copied where the file system cannot link (FAT, some network shares); files Guidedog did not make, such as a `CNAME`, are kept. Because the next output is a tree of its own, a file that becomes a folder, or a folder that becomes a file, is like any other change. Only a file Guidedog did not make that stands where the next output needs a folder, or below a path where it needs a file, stops the build: `A FILE IN THE OUTPUT IS IN THE WAY`, naming it, with the hint to move it or run `guidedog clean`. Nothing is published.
- Everything the journal will name is made durable in one pass (below).

- The journal (`.doctrees/<builder>.commit.json`, version 2) is saved through a flushed temporary file and a rename, and its folder flushed. It names the output folder, the next folder and its identity (device and inode), where the previous output goes in a two-rename switch (`.html.old`), and the records with their `.previous` links. Until it exists nothing published has changed; once it exists the commit is decided.
- The switch: where the system can, the output folder and the next are exchanged in one step (`renameat2` with `RENAME_EXCHANGE` on Linux, `renamex_np` with `RENAME_SWAP` on macOS; `host.exchange`), so a reader of the output folder, `guidedog` serve or a web server, sees the old tree or the new one and never neither. Elsewhere (Windows, file systems without the operation) it is two renames, output to `.html.old`, next to output. Then each record is renamed over its destination, the renamed folders are flushed, and the journal is removed. The `.previous` links go last.

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

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

The records must not move with the output. Default builder folders keep their records in the build folder’s `.doctrees`, outside the published tree. A custom output keeps its records in `.<output-name>.guidedog/.doctrees`, a stable sibling. This matters for `guidedog build -b html SOURCEDIR OUTPUTDIR`, where

putting records inside `OUTPUTDIR` would move the journal during the directory switch. Each custom target has its own manifest, so existing filenames in a different output cannot stand in for that target's published contents. Journal paths are absolute and resolve existing ancestors before naming a not-yet-created output, so recovery is independent of the next process's working directory and filesystem aliases. An output cannot contain the source directory. `lib/project/build_paths_test.odin` checks fresh and unchanged direct outputs, interrupted commits, destination isolation, and that source-layout refusal.

Cost: the spare output. Linking every file of the site into a fresh folder, and removing the replaced tree afterwards, costs two system calls per file for every publishing build. On an 1,800-document project (3,692 entries; Linux, ZFS, 32 cores), this made a one-page rebuild 0.62 s instead of 0.43 s, then 0.55 s with folder listings that open no entry (`host.list_folder`, from `readdir`'s names, types and inodes). So the replaced output is not removed: after an exchange it is already in the next folder, and after two renames it is moved there once the journal is gone. It is the spare output the next build writes into and completes, and it shares every unchanged file with the live output (they are hard links to the same files). Completing it compares the two trees folder by folder, by name, type, and inode from the listings: a name that is the same file as the live one is kept, one that differs is replaced by a link, one the live output does not have (and this build did not write) is removed. Nothing in the spare output is trusted: a stray file, a folder where a file goes, a file where a folder goes, or a file under a kept name that is not the live file is replaced or removed (`the_spare_output_never_leaks_into_the_published_one`). A producer that writes its own file (Typst) is given a path with nothing at it, so it can never write through a link into a published file; a folder a user made read-only in the output, once it is the spare output, is made writable again, since the spare is Guidedog's own. With the spare output, the same one-page rebuild is 0.43 s, as before the change (five runs: 0.43 s each after the first two, which fill the spare output once); a clean build and an unchanged build are unchanged (2.6 to 3.1 s and 0.33 s). The cost of a publication is listing the folders plus a link per changed file.

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

no journal and switches nothing, so an unchanged rebuild leaves the output folder, `environment.json`, and `<builder>.outputs.json` as they were, times and inodes included (`an_unchanged_build_rewrites_no_records`); a build that changes only records leaves the output folder in place.

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

Alternatives rejected.

- *Preflight planning of file moves.* Compute every move and removal, including removing a conflicting file or folder before creating a path, check each step's preconditions before the decision, and make each step idempotent and reversible. It keeps the live output being edited in place, so a reader sees a mixture while the moves run, and undoing a removal means keeping every removed file aside until the end: the same links a generation folder makes, without its single switch. Preconditions checked before the decision can still fail after it (a program opens a file, the disk fills). Rejected for the generation switch, which has one step to undo.
- *A symbolic link as the output folder,* switched with one rename. It makes `_build/html` a link, which some servers, archivers and Windows (without developer mode) handle badly, and changes what users see. Rejected.
- *Copying unchanged files* instead of linking. Correct, but it reads and writes the whole site for every change. Kept only as the fallback where linking fails.
- *Removing the replaced output after each switch.* Simple, but it costs a link and a removal per file per build (measured above). Replaced by the spare output.

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

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

Cross-document resolution happens between reading and writing, in the host: a pass over each document replaces toctrees with navigation lists and cross-

references with links, using the environment. Documents are still read and rendered by the allocation-free engine.

Project surface

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

- **Projects:** `Project`, `Load_Options`, `load(dir, options, registry)`, `unload`, and `CONFIG_FILE`. `Config` is public with the types of its fields (`Pdf_Document`, `Flag_Or_Name`, `Intersphinx_Project`, `Toml_Value`), since a caller may change the loaded configuration before a build, as `--typst` does.
- **Building:** `Builder_Kind` with `BUILDER_NAMES` and `builder_named` (the LaTeX builders' names map to pdf), `Build_Options`, `BUILD_BUDGET`, `build_limits` (for refusing options before a build), `build`, `Build_Result`, `release`, and `clean`.
- **Preview, translations, starting, migrating:** `Server` and `serve`; `Intl_Options`, `intl_update`, `intl_stat`, `Intl_File`, `Intl_Action`; `Quickstart_Options` and `quickstart`; `Migrate_Options`, `migrate`, `Migrate_Result`.
- **Paths:** `absolute` (against the working directory, without resolving a folder that does not exist yet) and `join`.

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

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

Directives and roles

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

- `toctree` (all options: `maxdepth`, `caption`, `name`, `hidden`, `includehidden`, `numbered`, `titlesonly`, `glob`, `reversed`), `only`, `index` and the `:index:` role, `glossary`, `productionlist`, `centered`, `hlist`, `seealso`, `rubric`, `versionadded`, `versionchanged`, `deprecated`, `versionremoved`, `sectionauthor`, `moduleauthor`,

`codeauthor`, `tabularcolumns`, `highlight`, `code-block`, `sourcecode`, `code` with Sphinx options (`linenos`, `lineno-start`, `emphasize-lines`, `caption`, `name`, `dedent`, `force`), `literalinclude` (every option), `math` with `label` and `nowrap`, `default-domain`, `default-role`, `todo` and `todolist`, `ifconfig`, and the figure and table numbering options.

- Roles: `doc`, `ref`, `numref`, `any`, `term`, `keyword`, `option`, `envvar`, `token`, `download`, `abbr`, `command`, `dfn`, `file`, `guilabel`, `kbd`, `mailheader`, `makevar`, `manpage`, `menuselection`, `mimetype`, `newsgroup`, `program`, `regexp`, `samp`, `pep`, `rfc`, `math`, `eq`, `math:numref`, and `extlinks` roles.
- Domains: `py` (`module`, `currentmodule`, `function`, `data`, `exception`, `class`, `attribute`, `property`, `method`, `staticmethod`, `classmethod`, `decorator`, `decoratormethod`, `type`), `c`, `cpp`, `js`, `rst` (`directive`, `directive:option`, `role`), and `std` (`program`, `option`, `cmdoption`, `envvar`, `describe`, `object`, `confval`, `term`, `label`), with their roles. Python signatures are parsed into name, parameters, defaults, annotations, and return type; C and C++ declarations are tokenized to find the declared name, and are shown as written.

Themes, templates, and pages

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

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

Code highlighting

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

Reading reStructuredText completely

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

Markdown

Markdown is first-class in Guidedoc's CommonMark reader; there is no separate reader and no plugin. Project `.md` documents support the native MyST syntax listed below. The pinned `myst-parser 0.16.1` fixtures provide comparison data, with recorded differences; they do not promise that every MyST project builds unchanged. Core syntax includes: directive fences (`{name}`) with options, roles (`{role}` followed by backquoted content), targets (`(label)=`), comments, block breaks, front matter, footnotes, tables, and cross-references. Its optional syntax is enabled as in Sphinx, with `myst_enable_extensions: amsmath, colon_fence, deflist, dollarmath, fieldlist, html_admonition, html_image, linkify, replacements, smartquotes, substitution, and tasklist`, with their `myst_*` settings. `extensions = ["myst_parser"]` is accepted and needs nothing, because the support is built in. Directives and roles are the same implementations reStructuredText uses, plugged into the reader through a hook, so a Markdown page produces the same document tree, pages, and book as its reStructuredText equivalent. A strict mode keeps plain CommonMark conformance for `guidedog convert`. `myst-parser 0.16.1`'s test fixtures, used as data, measure the rest.

Graphs

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

Typst

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

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

Intersphinx

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

Rationale and alternatives

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

Security and operational considerations

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

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

Backwards compatibility

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

Acceptance criteria and implementation status

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

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

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

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

Open questions

The plugin interface is left to a later record.

Review history

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

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

30 September 2026, direction: Guidedog is not a drop-in replacement. The earlier wording aspired to everything Sphinx does. On 11 October the maintainer clarified the target: an independent modern documentation system, designed idiomatically in Odin, with no Python execution and no goal of complete Sphinx compatibility. Sphinx is inspiration, not the specification. Comparisons measure useful authoring behavior and differences.

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

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

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

References

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

Lifecycle event

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

Lifecycle event

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

Lifecycle event

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