

---

# Memory budgets and what a build may read

Design, rationale, and review

---

## 01 / SOURCE

Write with meaning.

---

## 02 / STRUCTURE

Make relations clear.

---

## 03 / PUBLICATION

Read on screen and paper.

*Memory budgets and what a build may read*

Version 0.2.0

© 2026, Vikrant Rathore

Oct 11, 2026

---

# Contents

|                                                         |    |
|---------------------------------------------------------|----|
| Abstract . . . . .                                      | 1  |
| Motivation and scope . . . . .                          | 1  |
| Specification . . . . .                                 | 2  |
| Rationale and alternatives . . . . .                    | 7  |
| Security and operational considerations . . . . .       | 8  |
| Backwards compatibility . . . . .                       | 8  |
| Acceptance criteria and implementation status . . . . . | 8  |
| Open questions . . . . .                                | 9  |
| Review history . . . . .                                | 10 |
| References . . . . .                                    | 10 |



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

## Abstract

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

## Motivation and scope

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

In scope: the host’s budget (`lib/host`), the project build (`lib/project`), graphdog’s file access (`lib/graphdog`), and the Typst adapter’s contract

(lib/typst). Out of scope: time limits, and a worker process for Typst (GDS 0002, first open decision).

## Specification

### Checked sizes

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

### One budget, reserved before allocation

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

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

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

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

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

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

## Bookkeeping and writing, metered

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

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

- Catalogs: a PO or MO file is read within the budget and freed once parsed; the catalog, in an arena of its own, is charged its size once parsed, and each document's messages the gettext builder extracts are charged as its template grows.

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

### Refusal before allocation

Code behind a meter must handle a failed allocation, and every library it calls must survive one; each was checked by refusing, one test run at a time, every allocation it makes (the first, the second, ... and each from there on), and by a build that raises its budget from nothing to what it needs in small steps, so that the refusal falls in every metered step in turn (`builds_stop_cleanly_wherever_the_budget_ends`):

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

- Our own libraries behind meters (inventories, catalogs, the index and navigation pages, digests) check what they allocate; a catalog is compiled whole or not at all.

## When the budget is reached

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

- raise the limit to a new value, and the reservation is tried again;
- spill: working memory past the limit (sessions' storage, output, and source map, and what meters allocate) comes from the policy's `host.Disk` from then on;
- stop, and later refusals do not ask again.

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

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

## Working memory on disk

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

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

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

### Waiting without deadlock

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

### Outside the budget, and its bound

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

### What a build may read

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

- Graphs. `graphdog` gains `options.files`: after parsing and before layout it shows the caller every file name a graph gives through `image`, `shapefile`, `imagepath`, `fontpath`, and `<IMG SRC>` in HTML-like labels (`label`, `xlabel`, `headlabel`, `taillabel`), and sets the path the caller returns, or refuses the graph (`.File_Refused`). The build resolves a name against the document's folder, accepts a PNG, JPEG, GIF, or SVG file inside the readable folders whose path needs no escaping, and leaves `imagepath` and `fontpath` out. After rendering, each path in the drawing is replaced by the image as a `data: URL`, so pages and books show it without a copy and no path reaches the output; a drawing that would still show a path is refused. A graph that may name files is drawn every time rather than taken from the graph cache.
- Typst. The book's paths stay relative to `book_root`, the nearest folder holding the source folder, the include roots, and the graphs folder (no longer the

output folder, which Typst does not read). Typst runs in `typst_root`: a private temporary folder of links, at the same places, to only those folders, removed afterwards (the links, never what they lead to). A path in the book, its template, or its preamble then reaches those folders or nothing; the same holds for the embedded compiler, whose reads are relative to the root it is given. If the restricted root or any link cannot be made, the build stops with `typst.root` before compiling. It never falls back to the common ancestor. The hint explains temporary-directory permissions and Windows Developer Mode, or how to keep inputs inside one source root.

### –untrusted

Before discovery, `confine_untrusted` removes from the build’s configuration, each with a warning (`build.untrusted`), `include_roots` and every file or folder setting that leads outside the source folder: `html_static_path`, `html_extra_path`, `templates_path`, `locale_dirs`, `pdf_font_paths`, `autodoc_source_paths`, `odin_autoapi_dirs`, `html_logo`, `html_favicon`, `pdf_logo`, and `mathjax_config_path`. The build then reads only below the source folder. Ignoring with a warning was chosen over refusing the build, so a project from elsewhere still builds, visibly without what it cannot have.

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

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

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

### The Typst adapter’s contract

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

## Rationale and alternatives

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

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

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

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

Embedding a graph’s images, rather than copying them next to the drawing, makes one drawing serve pages and books alike, and keeps paths out of the output. Links for the Typst root, rather than copies, cost nothing for large image folders.

## Security and operational considerations

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

## Backwards compatibility

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

## Acceptance criteria and implementation status

Implemented, with tests: checked sizes and budget growth (`lib/host: sizes_never_overflow`, `growth_is_reserved_from_the_budget`,

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

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

Implemented, with tests, too: refusal before allocation and the policy (`lib/host: meters_charge_before_allocating` counts what reaches the backing allocator, `arenas_survive_refused_blocks`, `meters_prepay_a_bound`, `policies_raise_or_stop`, `spilled_memory_goes_to_disk`, `disk_memory_round_trips`, `disk_memory_is_bounded`, under `-sanitize:address` as well; `lib/jinja: refused_requests_never_crash_a_render`, `large_values_are_refused_before_they_are_built`; `lib/odindoc: parse_bounds_hold`, `refused_bounds_leave_files_out`; `lib/inventory: refused_memory_never_crashes_a_load`; `lib/gettext: refused_memory_never_crashes_writing`; `lib/project: builds_stop_cleanly_wherever_the_budget_ends` and `large_template_values_are_refused_before_allocation`, the review's case, whose budget never holds more than 4 MiB; `lib/cli`: the questions with a scripted terminal, and the flags; `tests/memory_test.odin`: the command line without a terminal). The review's reproduction in the September review peaked at 13 MB of resident memory, against 241 MiB before. The October candidate measurement is recorded in the beta review.

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

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

## Open questions

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

## Review history

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

## References

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

### *Lifecycle event*

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

### *Lifecycle event*

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

### *Lifecycle event*

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