Delroy

An agent harness built the slow way

ENGINEERING & ARCHITECTURE DOCUMENT

Version 1.1 · 10 August 2026

Repository state: `main` @ 758c42e

All figures independently verified against the working tree

226,999	4,315	201	405
LINES OF CODE	AUTOMATED TESTS	API ENDPOINTS	COMMITS SINCE 18 JUN 2026

Contents

1. Overview

> Delroy is an agent harness: the runtime, policy and workflow machinery that turns a model into an agent which can be handed real work. The model is the interchangeable part. The harness is the product.

Most agent tools are a chat box with a tool loop bolted to it. You type, a model answers, it reads a few files, and if it goes wrong you start over. That design has a ceiling, and everyone who has pushed against it has hit the same wall: one agent, one context window, one shot. Give it something genuinely large and it reads until it runs out of room, then hands back a half-finished map and a confident summary.

Raising that ceiling is not a prompting problem and it is not a model problem. It is a harness problem. A model that can call tools is not yet an agent you would leave alone with a repository; what stands between those two things is machinery — a loop that can run many steps without losing the thread, a tool executor that can refuse, a policy layer that decides what a turn may do *before* it does it, budgets that bound cost at every level from one turn to a whole run, and a way to measure whether any change to that machinery made the agent better or worse.

Delroy is an attempt to build that machinery properly — and, more unusually, to build it so the result can be audited rather than admired.

Concretely, the harness is: a streaming, multi-step, tool-calling loop; a bounded tool surface with an executor that enforces ownership and permission; delegation in which an entire tree of sub-agents draws on one shared allowance; multi-stage workflows whose shape and width the run itself decides; per-turn, per-stage, per-lane and per-run budgets; and a graded evaluation suite so that changes to prompts, truncation, compaction or routing land with numbers instead of impressions.

Four surfaces — a native desktop shell, a web workspace, a command-line interface, and a companion application that renders onto a pair of Even G2 smart glasses — are four ways into that one runtime, not four products. Delroy runs on your own machine and speaks to seven model providers, local and hosted, through a single request path. That local execution model is a real property with real consequences, covered in Section 4.5, but it is a deployment characteristic rather than the thesis.

1.1 The two bets

The project rests on two claims about the harness — not about the model — and both are legible in the code rather than asserted in a README.

The run should decide its own shape. Not a fixed pipeline. Not one agent with a bigger context window. A model drafts the workflow's semantics; a pure function forces its legality; a stage that has actually read the codebase decides how wide the next stage runs and who staffs each lane; and the tool executor refuses any write outside the lane that owns that path.

Method is the product. Guards go in the module that both callers import. Fixes are verified by deliberately breaking them. Architectures are retired by measurement rather than by taste. Refutations are written down beside confirmations. When a fix turns out to have caused the next failure, that goes in the document too.

Neither bet is finished. Both are tested, measured, and written down.

1.2 What a harness has to get right

The rest of this document is organized around the responsibilities a harness cannot delegate to the model. Each is a place where a plausible-looking implementation fails in production, and each has its own section.

RESPONSIBILITY	THE QUESTION IT ANSWERS	SECTION
Effort	How hard should this try, and what may that cost?	3
The loop	How does a turn run many steps without losing the thread?	5
Tools	What can the agent reach, and what refuses it?	5.1
Delegation	How do sub-agents share one budget rather than each getting a fresh one?	5.2
Context	How do budgets stay correct per model without a tokenizer?	5.3
Shape	Who designs the workflow, and what makes it legal?	6
Concurrency	How do several agents write at once without destroying each other's work?	7
Authority	What may a turn do, decided before it does it?	8
Observability	Is the telemetry good enough to retire an architecture?	9
Evidence	Do changes to the harness land with numbers?	10.5

1.3 How to read this document

Sections 2–4 describe what Delroy is and how it is put together. Sections 5–8 cover the four harness subsystems where the engineering is genuinely non-obvious: the agent runtime, the workflow engine, parallel execution, and the safety architecture. Sections 9–11 cover method, surface area, and open problems. The appendices carry the raw verification output, a module map, and the API and data-model inventories. Every quantitative claim here was measured against the working tree during preparation; Appendix A reproduces the commands and their output.

2. At a glance

2.1 Verified metrics

MEASURE	VALUE
Total tracked source	226,999 lines (Python, TypeScript, JavaScript, HTML, CSS; excludes vendored code)
Core Python	93,695 lines across 72 modules
Test code	58,953 lines across 134 pytest files
Python tests	3,454 passed · 1 skipped · 638 subtests · **0 failed, 0 warnings**
Web-client tests	370 passed across 44 files
Glasses tests	486 passed · 4 skipped across 41 files
Total tests	**4,315**
HTTP API	201 endpoints (103 POST, 74 GET, 12 PUT, 12 DELETE)
Persistence	21 SQLite tables
Model providers	7
Built-in agent tools	13
Permission modes	5
Effort levels	5
Graded evaluation tasks	38, in 6 task classes
Shipped core agents	6
MCP tool servers	7
Commit history	405 commits, 18 June – 10 August 2026

2.2 Composition

COMPONENT	SCALE
`client/server.py` — HTTP surface, routes, session and run management	25,703 lines
`agent.py` — agent discovery, deployment, project wiring, CLI core	6,813 lines
`client/pipeline.py` — workflow model, stage orchestration, fan-out	6,726 lines
`client/agent_runtime.py` — the agent loop and tool executor	5,836 lines
`client/pipeline_v3.py` — graph model, n8n interchange	3,318 lines

COMPONENT	SCALE
`client/effort.py` — the effort ladder, workflow materialization	2,120 lines
`client/run_policy.py` — permission modes, capability filters, gating	1,229 lines
Web client (`app-*.js` , 6 modules + support)	~24,000 lines
Glasses application (TypeScript)	59 modules, 16 screens

The distribution is itself a statement about where the work is. The four modules that constitute the harness proper — the loop, the policy layer, the workflow engine and the effort ladder — total roughly 15,900 lines, against a 25,700-line server whose job is largely to expose them over HTTP.

2.3 Documentation discipline

Of 72 core Python modules, **68 open with a module docstring**, with a median length of 850 characters. These are not one-line summaries. The prevailing convention is that a module states what it is *for*, what it must never import, and *why* — with the reason attached to the constraint.

> **REPRESENTATIVE — `CLIENT/LANE_OWNERSHIP.PY`**
>
> "This module is the vocabulary, and it lives on its own for a layering reason. The *partitioning* side (`client.pipeline` , which decides who owns what) and the *enforcement* side (`client.agent_runtime` , which refuses a write) must normalise a path identically — a set built with one spelling and checked with another is a guard that silently passes. `pipeline` imports no client module and `agent_runtime` does not import `pipeline` , so neither can host this; a leaf below both can, and both import it."

That paragraph is doing real work. It explains a file's existence, names the failure it prevents, and states the import rule that makes the prevention structural rather than conventional. This pattern recurs throughout the codebase and is the single strongest signal of how the project is built.

3. The organizing idea: the effort ladder

Almost every agent tool asks you to choose a model. Delroy asks a different question — *how hard should this try?* — and derives everything else from the answer.

```
light      low reasoning, a single pass
standard   medium reasoning, a single pass      (the default)
deep       high reasoning, a single pass
ultra      high reasoning AND a multi-stage workflow
max        ultra, plus stages that size themselves from what they find
```

Five levels, one meaning, on every surface: the chat composer, a backlog task, a scheduled automation, a voice command spoken into the glasses.

3.1 What a level actually controls

The level is not a label. It resolves to a concrete set of budgets, verified in `client/effort.py`:

LEVEL	REASONING	TURN-LIMIT SCALE	WALL-CLOCK CEILING	MAX STAGE EXECUTIONS	RUN CHARACTER CEILING
light	low	×0.5	20 min	12	uncapped
standard	medium	×1.0	45 min	12	uncapped
deep	high	×1.5	90 min	12	uncapped
ultra	high	×2.0	180 min	26	uncapped
max	high	×2.0	300 min	40	100,000,000

Three design decisions are embedded in that table.

The scale is a multiplier, not a replacement. The chat settings page owns the baseline turn limit; effort grades it. A flat override would make this module a second writer of a setting the user can see and edit — silently discarding a configured value.

The wall-clock ceiling exists because iteration counts do not bound time. `max_total_iterations` bounds stage *executions*; rework loops re-run stages. Without a clock, nothing bounds what a run costs.

The character ceiling exists only where fan-out exists. Every other budget in the system is per-turn, and a fanned stage turns each of those into a per-*lane* budget — twelve lanes each drawing a full allowance, with the run total bounded only by multiplication. The comment in the source records the measurements that set the number: 108.8M characters in one measured Max run (with two runaway lanes at 38.0M and 25.7M) before per-turn walls were tightened; 41–48M after. The ceiling sits above the healthy range and below the pathological one.

3.2 One rule, not a lookup table

Exactly one property distinguishes the top two levels: they build a pipeline. That is expressed as a named predicate — `runs_workflow` — rather than as `level == "ultra"` scattered across call sites.

> **WHY THIS MATTERS**
>
> Keeping it a rule rather than a per-surface table is what stops "does this surface run stages?" from becoming a question anyone has to look up. The rule is stated once, in the module docstring, and enforced in one place.

The module is also explicit about what it may not import: `client.pipeline` for one dataclass, `client.logging_setup` for a logger, and nothing else. It must never import `server.py`, so the levels stay trivially unit-testable and stage templates can be asserted without standing up an application. It must not import `client.task_intelligence` either — because that module imports *this* one, and the dependency cannot be allowed to reverse.

3.3 Level and shape are different questions

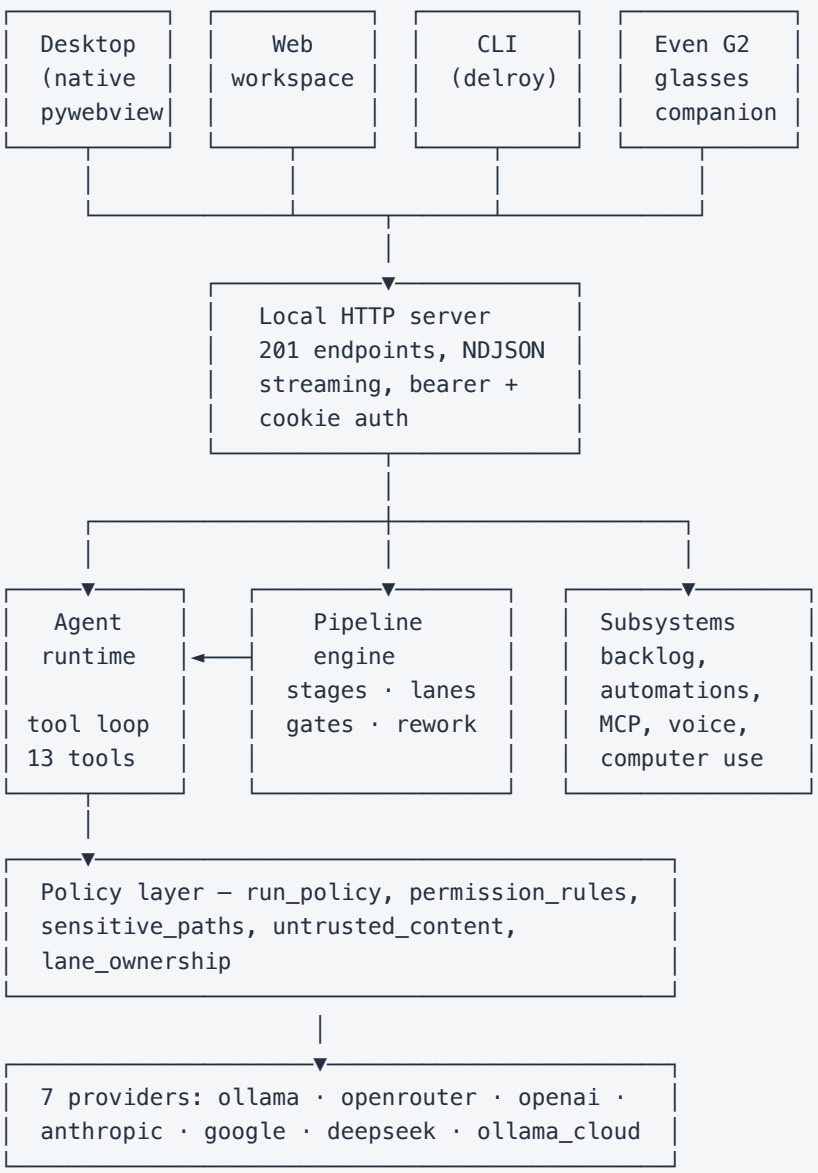
The level says how hard to try. The *shape* says what kind of work this is — and for a long time `ultra` silently answered the second question with "a code change."

The consequence was specific and instructive: asking for a security audit produced a planning stage planning an implementation nobody wanted, and a terminal gate reviewing a diff that did not exist.

The fix separated the two. `materialize_workflow_plan` turns a model-drafted plan into stages; `classify_shape` picks a curated skeleton when no model answered. Both end at the same linter.

4. System architecture

4.1 Four surfaces, one runtime



4.2 The layering rules

The architecture's most distinctive property is that its layering rules are written into the modules themselves, with reasons, and the import graph obeys them.

RULE	STATED IN	REASON GIVEN
The policy layer may never import the runtime or the server	`run_policy.py`	No circular dependency; policies stay trivially unit-testable

RULE	STATED IN	REASON GIVEN
The effort module imports one dataclass and nothing else of ours	`effort.py`	Levels testable without standing up an app
The agent runtime imports nothing from the server	`agent_runtime.py`	Dependency injection — the caller supplies the completion callable and the tool dispatcher
Ownership vocabulary sits below both its callers	`lane_ownership.py`	Partitioning and enforcement must normalize paths identically
The credential blocklist lives in a module both file-tool copies import	`sensitive_paths.py`	A guard inside one of two executors protects only one path

That last rule was learned rather than designed, and Section 8.2 tells the story.

4.3 Dependency injection as a testability strategy

`agent_runtime.run_agent_turn` receives its LLM completion callable and its extra-tool dispatcher from the caller. It therefore imports nothing from `server.py`. The same discipline applies to the pipeline engine: `execute_pipeline` takes its per-stage agent runner as an injected `run_subtask`, which makes it a pure generator of NDJSON-ready events.

The practical result is that the two most complex subsystems in the project can both be driven from a stub. That is why 133 test files can cover the agent loop, stage routing, cancellation, budget exhaustion, and rework semantics without ever starting a server or spending a provider credit.

4.4 Persistence

State lives in SQLite — 21 tables spanning sessions and messages, pipeline runs with per-stage and per-event history, background tasks, the cross-project backlog, scheduled triggers, and the workflow subsystem's versions, checkpoints, activations, pins, memory and idempotency records.

> **A REPRESENTATIVE BUG CLASS**
>
> An earlier round discovered that using `with` on a SQLite connection commits the transaction but never closes the connection. Fixing that single idiom exposed four further real bugs that the leaked handles had been masking — a lost update in the engine, a shutdown path that never joined its watcher threads, promotion logic clobbering a mid-build plan, and an engine started in `create_app` but stopped only in the lifespan handler.

4.5 Deployment: the harness runs where the code does

Delroy has no cloud component. The server binds locally, the desktop shell is a native window onto it, and nothing leaves the machine except the model call itself.

This is a deployment property rather than the project's thesis, but it is not incidental — it follows from what a harness has to do. The tool executor works against a real checkout, `run_shell` and `run_tests` invoke real processes, the language-server integration attaches to real files, and computer use drives a real desktop. A harness that enforces file ownership across concurrent lanes (Section 7) is enforcing it against one working tree that actually exists. Running beside the code is what makes those guarantees checkable rather than notional.

Two consequences worth stating plainly. Authentication is real despite being local: bearer tokens for programmatic clients, a cookie handoff for the native shell, and a cross-site check whose every failure mode is fail-closed (Section 10.6, item 2). And provider choice stays open — seven providers, local and hosted, behind one request path — so "local execution" constrains where the harness runs, not which model it may call.

`client/agent_runtime.py` is the streaming, multi-step, tool-calling loop that executes one agent's turn. It assembles four tool sources:

1. **A built-in development toolset** — 13 tools, scoped to the project and guarded by the agent's safety configuration.
2. **The agent's own MCP servers** — filesystem, git, and others, via `McpSession`.
3. **Extra application tools injected by the caller** — computer use, web search.
4. **Delegation** — when the caller opts in, a `delegate_subtask` tool that runs a fresh sub-agent turn in the same project, read-only by default, returning only its final report.

5.1 The built-in toolset

TOOL	CLASS
`read_file`, `list_directory`, `glob`, `grep`, `git`, `task_output`, `diagnostics`	read-only
`write_file`, `edit_file`, `multi_edit`, `notebook_edit`	mutating — approval-gated
`run_shell`, `run_tests`	executing — approval-gated

The read-only set is not an annotation; it is a named frozen set (`READ_ONLY_DEV_TOOL_NAMES`) that the read and plan permission modes intersect against. A tool's classification is therefore a fact the policy layer can compute, not a property each call site must remember.

5.2 Delegation is a budget, not a fact

Delegation depth is `MAX_DELEGATION_DEPTH = 1` — raised to 2 only by `max`-level runs. A run that raises it also passes a `DelegationLedger`, so the entire delegation tree draws on one shared allowance rather than each branch receiving a fresh budget.

The distinction matters: a per-branch budget makes total spend a function of tree width, which is exactly the multiplication problem Section 3.1 describes for lanes.

5.3 Context management

`client/context_engine.py` makes the loop's character-based budgets token-correct per model, without taking a tokenizer dependency:

- `TokenCalibrator` learns each model's real tokens-per-character ratio from the usage numbers providers already return — observation rides completions the loop already made, costing no extra calls.
- `context_window_for` maps a model id to its window from a static table plus user configuration, *never* from model output.
- `token_budget_chars` converts "fit the window with headroom" into the character budget the existing compaction machinery consumes.

The design constraint here is worth naming: the loop budgets in characters because characters are what it can measure without a tokenizer. Rather than introducing a tokenizer dependency across seven providers, the project made the existing measure correct.

5.4 The ledger and the handoff brief

A workflow's stages communicate through a ledger. The ledger itself stays complete — templates and run history need full artifact values — but the *prompt-side rendering* is bounded: the four most recent stages appear verbatim, and older ones condense to one line each.

The reason is stated in the source: a looping pipeline would otherwise drown its late stages in the history of superseded rework attempts.

The handoff brief has its own bounds, and one of them is deliberately generous:

> **FROM THE SOURCE**
>
> "A file list is the cheapest thing a stage can be told and the most expensive thing for it to rediscover — a measured `integrate` spent 201 `read_file` calls doing so — so it is generous."

`HANDOFF_FILE_LIMIT` is 120 entries. That number is not a guess; it is a response to a measurement.

6. Workflows: the model drafts, a pure function enforces

At the top two effort levels, Delroy stops running one agent and starts running a *workflow*: an ordered list of stages, each one agent doing one job, each able to read everything prior stages produced, the last one always a gate that decides whether the run passed.

The interesting question is who designs it.

6.1 Semantics from the model, legality from code

A routing prompt — shipped as an editable text file rather than as code, so its wording can be tuned without a release — asks the model to shape the workflow around what is actually being asked. Not from a menu:

- Work that **changes** code needs planning, the change, a stage that runs the tests, and a review.
- Work that only **looks** — an audit, a security review, a question to answer — must have *no* implementation stage and *no* test stage. There is nothing to build and nothing to run. It surveys, analyses, reports.
- Work that asks **why** reproduces the problem first, then explains it. No fix stage unless a fix was asked for.

The model's draft then passes through a materializer and a linter, both pure functions. They cannot be talked into an illegal shape:

- Between two and eight stages.
- Gates never receive parallel lanes.
- The last stage is always a gate.
- The gate is staffed by a reviewer, not by whoever did the work — nobody marks their own homework.
- A stage may only nominate an agent that is actually installed.

That final constraint carries more weight than it appears to. Membership in the installed set *is* the entire validation: a hallucinated agent name never reaches the registry at all.

6.2 Width discovered at runtime

The `max` level adds the property that makes it more than "ultra with a bigger budget": **a stage's width comes from the run, not from the draft.**

A drafted parallel split must be decided by a classifier that has never seen the codebase, so it can only split by *kind* — "authentication", "input handling". A stage that first goes and counts the actual routers can split by the actual work.

A stage opts in by naming an artifact key an earlier stage will publish. That earlier stage — one that has genuinely read the repository — produces a work-list, and the later stage fans into one concurrent lane per item.

The publishing stage may also nominate *which agent staffs each lane*. The rationale is precise:

> **FROM `WIDEN_STAGE_EXPLAINED`**
>
> "The stage that publishes the work-list has read the code, so it is the only party that knows a slice is a rendering problem rather than a persistence one — the drafter that wrote the stage never saw the repo. A nomination is accepted only when it names an agent in `installed`… a name is used rather than an index because the publishing stage and this function are separated by execution time — an index would silently resolve to a *different real agent* if the roster ever shifted, which is worse than refusing."

6.3 Every rule is a refusal

`widen_stage_explained` is built entirely from refusals, for a stated reason: **a fan-out the model got wrong costs N times as much as no fan-out at all.**

A missing artifact key, a non-list, an entry without a focus, fewer than two usable lanes, a gate, a stage that already carries a drafted partition, or a stage with more than one agent to choose from — each returns the stage *unchanged*, so it runs exactly as it would have without the function existing.

The function is also pure: no I/O, no model call, and no mutation. `config` is a mutable dict on a frozen dataclass and the stage map aliases the pipeline's own objects, so editing in place would write a discovered width back into the stored definition and desynchronize its fingerprint. The copy carries a fresh dict for exactly that reason.

6.4 Refusing beats truncating

When the work-list is larger than the lane cap, the widener **refuses to fan out at all** rather than truncating the list.

> **FROM THE SOURCE**
>
> "Truncating a READING work-list costs coverage. Truncating a WRITING one means the dropped entries' files are written by nobody, while the stage still reports success — the run then fails at the build with no trace of what went missing. Refuse instead: one agent doing all of it is slower and correct."

The same comment records why the cap is now a genuine ceiling rather than the concurrency limit wearing a second hat: they used to be the same number, so an eleven-piece plan against eight parallel slots did not run eleven lanes eight at a time — it ran **one** agent over all eleven. A measured run published exactly eight against a cap of exactly eight. One more piece and the entire fan-out would have vanished silently.

6.5 Refusals that explain themselves

Every refusal returns a reason string. That is a direct response to an operational failure:

> *"For three rounds they said nothing at all: a run whose `implement` collapsed to one agent looked identical to a run that never asked to fan out, and the difference had to be reconstructed days later by inferring stage shape from turn sizes."*

The reason is empty when the stage was widened — and also when the stage never asked to be, which is the overwhelmingly common case and must stay silent.

7. Parallel execution: ownership instead of isolation

The hardest unsolved problem in multi-agent coding is letting several agents *write* at once. Delroy has now been through both known answers, and the second is what the project is actively building.

7.1 The approach that was built, shipped, and deleted

The obvious answer is to give each lane its own git worktree. It was built. Each lane got its own checkout, wrote its files, committed to its own branch, and a merge step reassembled the work. It shipped behind a flag.

It failed in a specific and instructive way. A lane that needed a sibling's class in order to compile *could not see it*, so it wrote its own stub. Two lanes created the same path with genuinely different APIs. Git reported an add/add conflict, the merge rolled back, and the entire run's work was discarded.

In one measured run, **44% of the total spend occurred *after* that first merge failed**, and produced exactly one file.

No repair was available. Preferring either side of the conflict silently breaks the other lane's code, and no marker reliably identifies a stub as a stub. Prevention was the only option, so the architecture was retired.

7.2 The replacement

One shared working tree plus executor-enforced file ownership. No branches, no merge, no conflicts, no rollback. Each lane declares the exact paths it may create or modify, and the tool executor refuses everything else.

A lane that needs a sibling's class opens the sibling's *real file* and reads it — which fixes the stub problem at its root rather than repairing its symptom.

7.3 Where the engineering actually lives

The guard's details are the substance of the design.

Two fields, not one optional field. Deriving the restriction from "a path list, or nothing" is fail-*open*: lose the plumbing anywhere along the chain and you get nothing, which reads as unrestricted. The scope is therefore a separate field from the path set. A lost list denies everything. An empty set is not the same as an absent one.

The guard never sees the model's raw string. It inspects only the already-resolved, already-contained path. That makes the entire escape class — parent-directory traversal, absolute paths, symlinks pointing outward — structurally unreachable rather than merely filtered. A symlink pointing *at* a sibling's file resolves to the sibling and is refused correctly.

Path keys are Unicode-normalized and case-folded on case-insensitive volumes. Case-insensitivity is detected by *probing the filesystem*, not by checking the platform name:

> FROM `_DETECT_CASE_INSENSITIVE_FS`
>
> "macOS is usually case-insensitive (APFS) but case-SENSITIVE volumes exist, and a case-insensitive exFAT volume can be mounted under Linux. Guessing from the platform is wrong in both directions."

The normalization function carries a comment recording a real bug: the path is stripped of leading `./` with a `while` loop, *not* `lstrip("./")` — because `lstrip` operates on a character set and once turned `../x` into a valid claim on `x`.

Glob patterns are refused, not interpreted. Concrete paths are what make "do these two lanes overlap?" a decidable question. With globs it is not, and an undecidable overlap check is exactly how two lanes come to own one file.

Overlap detection checks three shapes, not one. File against file is the obvious case. The other two are a file inside another lane's directory, and one directory inside another's. A partition that misses those hands two lanes the same subtree and reintroduces precisely the last-writer-wins race that ownership exists to remove.

Half-owned partitions are rejected. `lanes_are_owned` returns true only when there are at least two lanes, every one declares at least one claimable path, and none overlap. One unowned lane in the set is one lane writing wherever it likes, beside lanes that cannot defend their files.

7.4 The shell is withheld, not policed

A writing lane does not get `run_shell` or `run_tests`. The reasoning is explicit and the cost is accepted:

> FROM `RUN_POLICY.PY`
>
> "The four edit tools resolve a path and are checked against the lane's owned set; `run_shell` and `run_tests` do not, and cannot be: `sed -i`, a redirect, or a codegen step reaches any file in the tree, and parsing shell for write intent is unbounded. So they are withheld rather than policed. `git` stays for reading only — the executor refuses its write subcommands for a scoped lane, because two lanes running `git add` concurrently corrupt the index for the whole run.
>
> The cost is real and deliberate: a writing lane cannot build or test itself. What replaces it is already there and is safer — `diagnostics` is read-only LSP, and every successful write already appends a diagnostics addendum. The build belongs to the single-agent stages that follow."

That is the shape of a good security decision: the unbounded problem is not solved, it is removed, and the compensating control is named alongside the cost.

8. Safety architecture

Delroy runs code on your machine. The permission architecture reflects that.

8.1 Five modes, with *ask* as the default

MODE	BEHAVIOR
`auto`	Full access. An explicit opt-in that the composer surfaces in yellow.
`ask`	Mutating tools pause for approval. **Default.**
`accept-edits`	Gates like `ask` but exempts file edits — "let it edit, ask before it runs things".
`read`	No mutation.
`plan`	Read-only exploration that must produce a reviewable plan.

Plan mode is enforced rather than suggested. An agent cannot submit a plan without having first either asked the user a question or explicitly declared the decisions it made unilaterally. The gate counts the cards actually drawn, not the intent.

Layered above the modes is a pattern rule engine (`permission_rules.py`) using Claude-Code-compatible strings — `run_shell(npm *)`, `git(push)`, `edit(src/**)`, `write(docs/**)`, or a bare tool name. Rules are evaluated at the approval gate *after* the mode and capability filters, which means they can never resurrect a tool the mode already removed. That ordering is the entire safety property.

8.2 One credential blocklist, shared by every file tool

`client/sensitive_paths.py` exists because of a specific, documented failure — and its docstring is the clearest single example of how this project records what it learns.

Two implementations of `read_file` / `write_file` / `edit_file` coexist: a disk-roaming computer-use copy, and a project-scoped development copy in the agent runtime. The chat dispatcher deliberately prefers the development copy for in-project paths.

The credential guard was originally built inside the computer-use copy — the branch handling the *less* dangerous case. A known in-project `.env` file, the single most likely credential store in any repository, went straight through to `read_text`.

> **THE LESSON, RECORDED IN THE SOURCE BECAUSE IT RECURS**
>
> "When one tool name has two executors, the guard belongs in a module both import, never in one of the executors."

The module also documents why it could not simply live in `run_policy`: it needs to resolve Delroy's own bearer token via `agent.delroy_data_root()`, and `run_policy` is documented as a lower layer that must not reach upward. The constraint was respected rather than quietly violated.

8.3 Untrusted content is fenced

Web pages, search results, MCP server output and on-screen text are wrapped in sentinel tags, and the system prompt tells the model once that anything inside the fence is *data to analyze*, never instructions to obey.

Two properties of the implementation are worth noting.

Forged markers are stripped first. `fence` neutralizes the sentinels case- and whitespace-insensitively, both closing *and* opening tags, so injected content cannot forge a fence boundary or spoof a `source` it did not come from.

Fixed sentinels are deliberate, not lazy. The attacker never sees the surrounding prompt, so the only breakout is emitting the application's own tags — which the stripping already handles. A nonce would add machinery without adding a property.

The module is candid about its own limits:

> "*This is an honest mitigation, not a guarantee: a determined injection can still influence a model. The value is that the boundary is explicit and any instruction found inside it is something the model should surface, not act on.*"

8.4 Shell and version-control classification

Every shell and version-control invocation is classified before it runs. One representative bug, found and fixed: a bare `git stash` has no sub-action, so it classified as a *read* — while the same command with its explicit sub-action classified as a *write*. The bare form bypassed all three boundaries.

That is the kind of defect that only surfaces when someone is deliberately hunting for the gap between a classifier's model of a command and the command's actual behavior.

9. Measurement over taste

This section describes the moment that best characterizes how the project is run.

9.1 The telemetry that retired an architecture

After four rounds of work on parallel implementation — budgets, landing behavior, merge semantics, conflict resolution — runs at the top effort level still produced disappointing results. The natural next move is another round of fixes.

Instead, a per-turn telemetry log kept *outside version control* (so it survives code reverts) was analyzed across 351 recorded turns. The findings reframed the problem entirely:

MEASUREMENT	VALUE
One representative run	33 turns, 48.3M characters, 52 minutes
Turns in that run that changed a single file	**8**
Across the last three runs, characters spent on turns that changed nothing	**82M of 131M**
Read-to-write ratio per turn	roughly **100:1** (~330k input tokens vs ~3k output)
Engineer agent: turns that wrote files	13 of 31
Test generator: turns that wrote files	**0 of 5**
Code reviewer: turns that wrote files	**0 of 3**

Lanes were burning their entire budget re-orienting themselves, then landing with a *report* instead of code.

The merge machinery was where the *bugs* lived. It was not why runs produced nothing.

> **RULE ADOPTED**
>
> **Cost and landing being fixed is not the same as results arriving. Measure files changed per turn, not characters spent.**

That measurement retired an entire architecture and selected its replacement. It also produced a downstream discovery no amount of code reading would have found: the staffing prompt was instructing the model to choose *whoever best understands the piece rather than whoever would implement it* — actively selecting advisors for lanes that would later be required to write.

9.2 Other measurement-driven findings

8% of turns burned 58% of total spend. Identified from a turn-level ledger, not from inspection.

Context compaction was thrashing on nearly every step — over 1,500 elisions across 200 steps — because a hysteresis watermark had been placed inside the wrong conditional, destroying the provider's prompt cache on each pass.

Raising the conversation budget to fix that would have cost 2.5× more, not less. This is the finding that most justifies the method: the intuitive fix was quantitatively backwards, and only measurement revealed it.

9.3 Falsification as the primary bug-finder

Every fix is verified by breaking it on purpose and confirming that a test fails. The discipline pays for itself in an unexpected way:

> RULE ADOPTED
>
> **A falsification that passes is a coverage gap, not a pass.**

Three separate wirings — a cancellation signal, a contract check, and a status flag forwarded across a module boundary — were each broken deliberately, and the entire suite stayed green. All three were untested seams. This single technique has now found five such seams across two rounds, and is the most reliable bug-finder in the project.

10. Engineering method and evidence

The phrase "not vibe-coded" deserves evidence rather than protest. This section is the evidence.

10.1 A green baseline that is defended, not declared

The Python suite runs **3,454 passed, 1 skipped, 638 subtests, 0 failed, 0 warnings** in 183 seconds. The web client runs 370 tests across 44 files, all green. The glasses application runs 486 tests across 40 files, with one file (4 tests) skipped by design — the opt-in live-transport suite, which requires a running server and real credentials. The single Python skip is an environment guard rather than a design one; Appendix A.1 records it and what it implies for the release gate.

Getting to zero warnings required discovering that **a green baseline is only as honest as its warning filters.** One hardening round closed with ten consecutive clean runs while 49 resource warnings fired the entire time — invisible, because a warning raised *during garbage collection*, which is exactly when an unclosed handle is reclaimed, gets downgraded by the test runner into a different, non-failing warning class.

The following round hunted all 49 to zero by bisecting the file list rather than filtering them, and adopted the rule: promoting resource warnings to errors is not by itself a leak gate, because garbage-collection-time warnings are downgraded — the release check must also assert the downgraded count is zero.

A second rule from the same round: **a clean-run count is not a flake gate.** Ten clean runs pass roughly 70% of the time against a 3–5% flake rate.

10.2 The pytest configuration is a design statement

```
[pytest]
filterwarnings =
    ignore:Using `httpx` with `starlette.testclient` is deprecated
    ignore:websockets.legacy is deprecated:DeprecationWarning
    ignore:websockets.server.WebSocketServerProtocol is deprecated:DeprecationWarning
```

Three filters, each matched on an exact message, each for a third-party deprecation the project cannot fix — with a comment explaining that the scoping exists so genuinely new deprecations from the project's own dependencies still surface. That is a meaningfully different artifact from a blanket `ignore::DeprecationWarning`.

10.3 Documented hardening rounds

The project has run multiple documented hardening rounds, each a multi-front audit in which every finding had to supply either a reproduction that was actually run, with real output, or an explicit instance argument naming concrete callers.

One round's ledger: **23 raw findings → 20 confirmed, 3 refuted, 16 deduplicated items.**

The refutations are the proof the pass was real rather than ceremonial:

- One "resource leak" turned out to be intended design — a session-scoped singleton owning the webhooks for every deployed workflow, where the proposed stop control would have killed all of them at once.
- One "race condition" named a method with zero callers, guarded by a lock it did not know about.
- One finding **fabricated the tool names it was about**; no such tools exist.

Each was written down as refuted rather than quietly dropped.

10.4 Corrections are recorded, including expensive ones

One round's planning document contains the sentence: "*an earlier draft of this document called it a production leak; that was wrong and is corrected here.*"

Another records: "*I misdiagnosed this. Building what I first described would have changed nothing about the run.*"

A third records that two of its own fixes were the direct cause of a later live failure — a per-turn limit had been scaled by two, and a workflow is many turns, so multiplying a per-turn ceiling multiplied the entire run's cost.

And one ordering rule that could only come from having gotten it wrong:

> **RULE ADOPTED**
>
> **Never ship a budget cut before the truncation-is-visible fix.** Halving a step limit before stages were able to report "I stopped early" would have converted the one stage that was actually working into a silent partial success that looked like a clean pass.

10.5 Competence is measured, not asserted

A graded evaluation harness runs **38 fixture tasks** across six task classes:

CLASS	COUNT	WHAT IT GRADES
`EvalTask`	16	Single-turn agent behavior
`PlanEvalTask`	8	Plan quality and contract adherence
`RouteEvalTask`	6	Routing decisions
`AutonomyEvalTask`	4	Desktop-autonomy behavior, including safety refusal
`PipelineEvalTask`	2	Full multi-stage pipeline runs
`ScaffoldEvalTask`	2	Agent scaffolding

The tasks are behavioral rather than academic: fix a failing test (the bug, not the test). Rename a constant. Answer a question about an unfamiliar codebase. Find a value buried in a huge file. Respect the stated scope. Admit what you do not know. Refuse to touch a protected file. Survive context compaction. Orient graph-first, and be honest when the graph is unavailable.

Each task seeds a throwaway project, builds the same system prompt a deployed agent would receive, runs one real turn, then applies programmatic checks and records pass/fail, tool calls, model calls, characters sent, and wall time.

The harness enforces a property that most eval suites lack:

> **FROM** `CLIENT/EVALS/TASKS.PY`
>
> "Checks must discriminate — they fail on the untouched fixture with an empty answer (`test_evals.py` enforces this), so a passing run means the agent actually did the work."

The stated purpose: so that changes to prompts, truncation, compaction, or routing land with numbers instead of vibes.

10.6 A residual-risk register with reopening triggers

The project maintains a register naming every accepted tradeoff, categorizing it — working as intended, documented fail-safe, or maintainability debt — and stating the specific trigger that should reopen it.

#	RESIDUAL	CATEGORY	TRIGGER TO REVISIT
1	Chat-router closure coupling	Maintainability debt	Chat routes need substantial edits
2	pywebview native `Sec-Fetch-Site`	Documented manual check (fail-closed)	A desktop-shell or auth change
3	Turn overrun ≤ one step	Documented tradeoff	A hard wall-clock SLA is introduced
4	`request_options` excluded from sub-turns	Working as intended	A sub-model must receive reasoning specs
5	One-shot MCP discovery-probe stderr	Bounded by timeout	Probe reports mysterious hangs
6	`server.py` size	Maintainability debt	A cohesive unit becomes extractable

Item 4 is annotated "**Do not 'fix' this**" — forwarding a reasoning spec to a delegated sub-model can return a 400 from a non-reasoning model, so the exclusion is deliberate and now locked by a named test.

The register says out loud that the main server module is too large and that two of its routes are over-coupled, with the phased extraction plan written down separately.

CURRENT STATE, MEASURED FOR THIS DOCUMENT

The register records `server.py` at approximately 17.7k lines. It now measures **25,703 lines** — the module has grown by roughly 45% since that entry was written. Item 6's trigger ("a cohesive unit becomes cleanly extractable") remains the right one, but the debt it tracks has increased materially and the entry is now understated.

That observation is included because a document of this kind is worth nothing if it only reports the figures that flatter. The register's own methodology is what makes the discrepancy findable.

10.7 A release checklist with exactly one manual gate

The checklist runs all three suites in one command, then requires a clean import, then an opt-in live-smoke run against a sanctioned model.

The single manual gate is the native desktop smoke test, and the reason it must stay manual is stated: it depends on which security header a real embedded browser engine emits, which cannot be unit-tested. Every failure mode of that check is documented as fail-closed — no header falls back to a loopback check and is allowed; a spurious one produces a loud 403 on first launch, never a silent authentication bypass.

The checklist also insists the opt-in live transport suite be run at least once per release, with the reason attached:

> *"A suite that has only ever skipped is not evidence of anything."*

11. Reach: what the harness is wired into

The harness is the spine. The surface area is what makes it reachable from wherever the work is actually happening — and several of these are harness features in their own right rather than conveniences bolted alongside one.

11.1 Agents as first-class, portable artifacts

Any agent can delegate to any other through one unified delegation tool, inheriting permission mode and approval gates. Agents can create, revise, and retool *other* agents. Six core agents ship with the project: `delroy-core`, `delroy-chat`, `delroy-architect`, `delroy-pipeline-architect`, `delroy-desktop`, and `delroy-voice`.

Routing prompts ship as editable Markdown — `agentic_route.md`, `context_switch.md`, `desktop_intent.md`, `run_synthesis.md`, `task_agent.md`, `workflow_shape.md` — so the semantics of routing can be tuned without a code release.

11.2 Backlog and autopilot

A cross-project to-do list with dependencies, a fail-closed judge, per-task effort selection, and capture-from-chat. Task effort is deliberately restricted to `deep`, `ultra`, and `max` at the point of hand creation:

> **FROM `EFFORT.PY`**
>
> "A task runs headless, so the question is not 'how fast do I want this back' — nobody is waiting at the screen — but how much of the work it should be trusted with unattended… The two cheapest levels are deliberately absent rather than removed. Auto can still land on them (a typo is a typo however it runs), and a task carrying one keeps it and shows it — but the complaint that started the effort ladder was tasks running one thin pass and achieving nothing, and a menu is where that mistake gets made by hand."

11.3 Automations

Scheduled prompts and workflows. A scheduled prompt is modeled as a synthetic single-stage pipeline rather than as a second execution path — one runtime, not two.

11.4 Model Context Protocol

Full MCP support with a local registry snapshot and a curated floor, searching in roughly twelve milliseconds. Seven MCP tool servers ship in `tools/`: the main Delroy server, agent administration, search, Playwright, and three computer-use variants.

11.5 Computer use

Real desktop control with its own driver layer (`computer_use_drivers.py`, 1,958 lines) and an audit log. The shared implementation is imported by both the MCP server and the chat-mode tool loop, so the tool surface, safety checks and audit log cannot drift apart between the two.

11.6 Workflow automation and n8n interoperability

Pipelines v3 defines a JSON-native graph model that is a deliberate superset of n8n's: stable node ids, typed and named ports, explicit connection kinds, credential *references* rather than values, and loss-aware interoperability metadata.

The execution adapter delegates node behavior to the official pinned n8n runtime rather than maintaining hundreds of approximate Python reimplementations, with no HTTP or API dependency — workflows and credential bindings are imported into an isolated local n8n data directory and executed by the pinned CLI.

11.7 Voice

A local voice stack — `whisper.cpp` for recognition, `piper` for speech — with spoken approvals, context-aware biasing, and a silence gate.

11.8 The Even G2 glasses application

A TypeScript companion app of 59 modules and 16 screens, delivering chat, effort selection, permission modes, task lists, live run mirroring, plan review and approval, and spoken answers to questions — all on a monochrome heads-up display.

The display constraint produced one of the most disciplined pieces of engineering in the repository. The G2 firmware font covers a small, undocumented subset of Unicode, and none of it can be checked from the desktop: the phone panel is a WebView using the *phone's* fonts, so a glyph the glasses cannot draw looks perfect in the preview. Three features shipped invisible this way.

On-glass probing established that exactly **four** non-ASCII characters draw:

```
U+25B6   U+00B7   U+25CF   U+25BC
```

and thirteen, probed in the same row on the same screen, do not:

```
U+25B8 U+2218 U+25C9 U+2726 U+2713 U+2717 U+2715
U+2699 U+276F U+2302 U+26A0 U+21C4 U+22EF
```

The finding recorded alongside the data is the valuable part:

> **FROM** `GLASSES/SRC/UI/GLYPHS.TS`
>
> "Note WHAT survived: large solid geometric shapes, plus one Latin-1 dot. The small or hollow variant of the very same shape did not — U+25B8 next to U+25B6, U+2218 and U+25C9 next to U+25CF. Never reason from 'the font has U+25CF' to 'the font has U+25C9'; it demonstrably does not."

Two independent layers enforce the rule, because one is not enough. `tests/glyphs.test.ts` parses every other module and fails the build on a non-ASCII string literal — catching what the team authors. And `safeText()` runs at the display boundary — catching what arrives from the server at runtime. One module is the only place in the application permitted to contain a non-ASCII character.

Cross-device mirroring is reference-counted, so stopping a run from your face stops it on your desk.

12. Roadmap and open problems

The near-term work is finishing what Section 7 describes: making parallel writing actually produce parallel results. The machinery is landed and passing. What remains is genuinely open, and is worth stating precisely rather than optimistically.

12.1 The model does not yet reliably publish file-level work-lists

A planning stage is asked to emit, per workstream, an identifier, a focus, and a complete list of files. When it emits nothing, the widener finds no work-list and the implementation stage falls back to a single agent — the machinery is correct and completely inert.

That demand is now a validated contract rather than a paragraph in a prompt, which routes a miss into the correction-turn machinery that already existed. If one re-ask still is not enough, the next lever is the wording of the demand itself: four required fields including a complete file list may simply be too much to ask for in a single artifact.

12.2 A live canary is owed

Offline tests can prove the partitioning, the enforcement, and the failure semantics. They cannot prove that a model will cooperate.

The pass criterion is a single number: **the share of lane turns that change at least one file, currently 24%.** If it does not rise, the machinery is fine and exactly one prompt needs changing.

12.3 Budgets need rebalancing against real workloads

In the best run to date, the survey, planning and implementation stages consumed the entire run budget between them, leaving nothing for integration, testing or QA.

That run *succeeded at implementation* — every one of the target project's 87 tests passing — and still recorded as failed, because an exhausted ledger let the graph keep walking: sixteen further stage executions each started, found nothing left to spend, reported themselves incomplete, and routed to rework.

That specific failure is fixed. The budget question behind it is not.

12.4 Structural work

Extracting the over-coupled chat routes along their documented plan, and bringing the main server module down in size. As Section 10.6 notes, this debt has grown rather than shrunk since it was first registered, which strengthens rather than weakens the case for the phased extraction already written down.

12.5 Evaluation

Pushing the evaluation suite from correctness toward judgment. The 38 current tasks grade whether the agent did the right thing; the harder and more valuable question is whether it chose the right thing to do.

13. Conclusion

Delroy is an agent harness whose distinguishing bet is that **the run should decide its own shape.** A model drafts the semantics, a pure materializer forces the legality, a stage that has actually read the code decides how wide the next stage goes and who staffs it, and the executor refuses any write outside the lane that owns the path.

The reason this document spends its length on loops, ledgers, budgets, refusals and ownership rather than on prompts is that those are where an agent stops being a demonstration and starts being something you can hand work to. The model is replaceable — Delroy speaks to seven of them through one path, and the harness does not change when you switch. What does not come free with any model is the machinery that bounds it, refuses it, measures it, and tells you honestly when it produced nothing.

The second bet is on method, and it is the one this document has spent the most space on, because it is the one that transfers. Guards go in the module both callers import. Fixes are verified by breaking them. Architectures are retired by measurement rather than by taste. Refutations get written down next to confirmations. And when a fix turns out to have caused the next failure, that goes in the document too.

The residual-risk register opens with the project's own assessment of its own history, and it is the most honest line anywhere in the repository:

> "*Seven hardening rounds took Delroy from vibe-coded to a client with a green baseline.*"

It started as one. It was not left as one.

Appendix A — Verification log

Every figure in this document was measured against the working tree at commit `758c42e`. The commands and their output follow.

A.1 Test suites

```
$ python3 -m pytest -q
3454 passed, 1 skipped, 638 subtests passed in 183.29s (0:03:03)
EXIT=0

$ cd client/static && npx vitest run
 Test Files  44 passed (44)
      Tests  370 passed (370)
   Duration  6.17s
EXIT=0

$ cd glasses && npx vitest run
 Test Files  40 passed | 1 skipped (41)
      Tests  486 passed | 4 skipped (490)
   Duration  3.46s
EXIT=0
```

Both skips are conditional rather than disabled tests, but for different reasons.

The glasses skip is by design: `tests/live.test.ts`, guarded by `describe.skipIf(!url || !token)`, which the release checklist requires be run deliberately at least once per release.

The single Python skip is environmental:

```
$ python3 -m pytest -q -rs
SKIPPED [1] client/tests/test_agent_runtime.py:160: ripgrep not installed
```

This is worth flagging rather than glossing. The release checklist's automated gate specifies **0 failed, 0 warnings, 0 skipped**, and the run above returns 1 skipped on a machine without `ripgrep` on its path. The test is correctly guarded — it exercises a tool the runtime uses only when available — but the checklist's zero-skip criterion is therefore host-dependent, and a release run must either install `ripgrep` or record the skip as accepted. On this measurement host the gate would need that decision.

A.2 API surface

```
$ grep -cE '@app\.(get|post|put|patch|delete|websocket)' client/server.py
201

$ grep -oE '@app\.(get|post|put|patch|delete)' client/server.py | sort | uniq -c
 103 @app.post
  74 @app.get
  12 @app.put
  12 @app.delete
```

A.3 Scale

```
$ git ls-files '*.py' '*.js' '*.ts' '*.html' '*.css' | grep -v vendor | xargs wc -l | tail -1
  226999 total

$ git ls-files '*.py' | grep -v '/tests/' | xargs wc -l | tail -1
   93695 total

$ git ls-files 'client/tests/*.py' | xargs wc -l | tail -1
   58953 total

$ git ls-files 'client/tests/*.py' | wc -l
     134

$ git rev-list --count HEAD
405
```

A.4 Configuration constants

```
$ python3 -c "from client import agent_runtime; print(len(agent_runtime.DEV_TOOL_NAMES))"
13

$ python3 -c "import providers; print(len(providers.PROVIDERS))"
7

$ grep -n 'PERMISSION_MODES' client/run_policy.py
32:PERMISSION_MODES = ("auto", "ask", "accept-edits", "read", "plan")

$ grep -n 'EFFORT_LEVELS' client/effort.py
44:EFFORT_LEVELS = ("light", "standard", "deep", "ultra", "max")
```

A.5 Documentation coverage

```
$ python3 measure_docstrings.py
core modules: 72   with module docstring: 68   without: 4
median module-docstring length: 850 chars
```

The four modules without a module-level docstring are `agent.py`, `client/server.py`, `client/voice.py`, and `tools/delroy_mcp.py` — in each case the largest or oldest file in its area.

Appendix B — Module map

B.1 Core Python modules by size

MODULE	LINES	RESPONSIBILITY
`client/server.py`	25,703	HTTP surface, routes, sessions, run management
`agent.py`	6,813	Agent discovery, deployment, project wiring
`client/pipeline.py`	6,726	Workflow model, stage orchestration, lane fan-out
`client/agent_runtime.py`	5,836	Agent loop, tool executor, delegation
`client/pipeline_v3.py`	3,318	Graph model, n8n interchange
`client/cli.py`	2,447	Command-line interface
`client/automation_store.py`	2,165	Scheduled automation persistence
`client/effort.py`	2,120	Effort ladder, workflow materialization
`agent_portability.py`	2,043	Agent export/import
`client/computer_use_drivers.py`	1,958	Desktop control drivers
`client/computer_use_tools.py`	1,771	Shared computer-use tool surface
`client/evals/tasks.py`	1,738	38 graded evaluation fixtures
`client/evals/harness.py`	1,621	Evaluation harness
`providers.py`	1,577	Seven-provider request path
`client/n8n_runtime.py`	1,577	Pinned local n8n execution adapter
`tools/delroy_mcp.py`	1,442	Primary MCP server
`client/voice.py`	1,317	Voice capture, recognition, synthesis
`client/run_policy.py`	1,229	Permission modes, tool gating
`client/architect_tools.py`	1,218	Pipeline authoring tools
`client/pipeline_plan.py`	1,045	Plan model

B.2 The policy layer

MODULE	LINES	ROLE
`client/run_policy.py`	1,229	Permission modes, capability filters, approval gating
`client/pipeline_plan_validation.py`	644	Plan legality

MODULE	LINES	ROLE
`client/permission_rules.py`	360	Pattern rules layered over mode policy
`client/retry_policy.py`	254	Retry semantics
`client/lane_ownership.py`	207	Path ownership vocabulary
`client/sensitive_paths.py`	126	Shared credential blocklist
`client/untrusted_content.py`	56	Injection fencing

The whole policy layer is 2,876 lines — about 3% of the core Python, and the part of the harness that decides what every turn is allowed to do. `untrusted_content.py` at 56 lines is the smallest module carrying a named security property in the project, which is the point: the boundary is small enough to read in one sitting.

B.3 Web client

MODULE	LINES
`styles.css`	11,496
`app-plugins.js`	6,095
`app-chat.js`	4,874
`vault-graph.js`	4,044
`app-core.js`	3,603
`app-desktop.js`	3,500
`app-vault.js`	3,155
`index.html`	1,856
`app-boot.js`	1,661
`tasks.js`	1,085
`autonomy-mini-graph.js`	1,060
`graphify-view.js`	836
`model-routing.js`	552

The six `app-*.js` modules are the product of a deliberate decomposition of a former single `app.js`. They are classic scripts sharing one global scope, loaded in a fixed order — `app-core` first, `app-boot` last — with function declarations hoisting within each file and cross-file references resolving at call time. That constraint is documented in the header of every one of the six.

B.4 Glasses application

59 TypeScript modules, of which 16 are screens: approval, chat, chats, effort, index, menu, mode, plan-approval, plan-view, project-sessions, projects, prompt-recorder, question, status, stop, tasks.

Supporting layers: `bridge/` (Even App bridge), `delroy/` (14 modules — HTTP, NDJSON streaming, sessions, turns, plans, questions, approvals, tasks, projects, voice, config), `graph/` (5 modules), `ui/` (router, layout, geometry, glyphs, input, live-view, and the screen set), `voice/` (capture, hold-capture, mic arbitration, WAV), `pairing/`.

Appendix C — API and data model

C.1 Endpoint families

The 201 endpoints group into these families:

FAMILY	PURPOSE
`/api/agents/*`	Agent discovery, capabilities, per-agent MCP servers, plugins, tool toggles
`/api/chat/*`	Sessions, messages, streaming, live tap, context, rewind, worktree, settings
`/api/pipelines/*`, `/api/pipeline-runs/*`, `/api/pipeline-plans/*`, `/api/pipeline-drafts/*`	Workflow definition, versioning, verification, run history, event streams
`/api/backlog/*`	Cross-project tasks, dependencies, autopilot, settings
`/api/triggers/*`	Scheduled automations
`/api/projects`, `/api/project/*`	Project registry and file access
`/api/providers/*`, `/api/models`, `/api/routing`, `/api/openrouter`	Model catalog and routing overrides
`/api/mcp/registry`, `/api/marketplace/*`	MCP registry snapshot, agent marketplace
`/api/n8n/*`	Credential types, node schemas, extensions, runtime
`/api/computer-use/*`	Desktop control and live models
`/api/voice/*`	Voice configuration and assets
`/api/glasses/pairing`	Glasses pairing payload
`/api/graphify/*`	Knowledge-graph build, query, status, sync events
`/api/settings`, `/api/appearance`, `/api/permissions`, `/api/secrets`	Configuration
`/api/health`, `/api/logs`, `/api/search`	Operations

C.2 Data model

21 SQLite tables:

GROUP	TABLES
Conversation	`sessions`, `messages`
Runs	`runs`, `pipeline_runs`, `pipeline_run_stages`, `pipeline_run_events`
Verification	`pipeline_verifications`, `pipeline_verification_events`
Backlog	`backlog_tasks`, `backlog_settings`
Automation	`triggers`, `background_tasks`
Workflow subsystem	`workflow_versions`, `workflow_activations`, `workflow_checkpoints`, `workflow_idempotency`, `workflow_memory`, `workflow_pins`
Configuration	`client_settings`

C.3 CLI surface

```
delroy agent | client | cli | clean | computer-use | current | desktop
       disable | doctor | enable | list | ollama | openrouter | pipeline
       providers | query | remove | reset | run | search | status
       trigger | use | vault
```

Prepared 10 August 2026. Figures are a point-in-time measurement of the working tree at commit 758c42e, not a release tag. All commands in Appendix A were executed during preparation; their output is reproduced verbatim.