Assay — the methodology

Work held true. To assay a metal is to test its purity rather than take the seller's word. Assay applies the same standard to agent-fleet software work: a toolkit and a set of conventions that put every claim of done behind a machine-checkable gate.

> **About this document.** It is generated from the pages of assay.guide — the same words, re-set for print — so the paper and the site cannot say different things. It is not a separately written paper. Two things on the site are not reproduced here: the flow **diagrams** (each appears below as its caption text only) and the site's navigation and quickstart sections. For those, and for the current version of everything below, read assay.guide.

Methodology

When agents do a large share of a team's implementation work, status becomes an unreliable narrator. Assay is a methodology for that regime: it restructures work into documents a script can check and routes every claim of done through a gate that does not belong to the author. This page is the overview — the discipline as a whole, and the reason each mechanism exists. Each mechanism has its own explainer: Briefs, Registers, Lifecycle, Desk roles, Model tiering, and statusgen.

The problem: the narrator grades its own exam

The controls that keep human-paced software work honest assume two things: the actor who says "done" is durably accountable, and status assertions arrive at human volume. A fleet of spawned agents breaks both at once. Assertions arrive at machine speed, and the asserter is a session that will not exist tomorrow to answer for them. Under that regime the failures are predictable, and each one names a mechanism Assay carries.

FAILURE	REPAIR	MECHANISM
Agents grade their own homework — "done" becomes a self-report, the weakest signal available.	A verification step that, by rule, cannot be performed by the author.	Lifecycle
Plans drift from reality — the intended state lives in prose nothing can check.	The plan is a typed document a linter parses.	Briefs
Prose dependencies break silently — "depends on the oracle work" survives the rename that broke it.	Typed references, so a broken one is a lint failure.	Briefs

FAILURE	REPAIR	MECHANISM
Priority collapses to proximity — each worker takes the next item in its own lane.	A computed queue that weighs every stream at once.	statusgen
The system's memory is editable by the ones being remembered.	Append-only registers whose deletions are machine-visible.	Registers

Issue trackers, CI, and code review are necessary, and Assay assumes them. But they were built for teams where the person who says "done" is accountable in a way a spawned agent is not. The gap they leave — consistency checking over agent-authored work-state, with verification gates keyed to risk — is the gap this methodology fills.

The design stance

Three commitments run through every mechanism.

- **Every rule is stated with its reason.** A convention whose rationale is lost decays into ritual, and a fleet of agents is very good at satisfying rituals literally. Each rule carries the failure it prevents, so it can be re-derived or retired on its merits when circumstances change.
- **Gates over trust.** A rule that depends on discipline is a wish; a rule the machinery enforces is a gate. Wherever a norm can become a check, a check a lint failure, and a lint failure a blocked merge, it does. The friction is the feature.
- **Claim the weaker, true thing.** The strong claim — that status is measured and agents cannot lie — is false: the sensors are writable by the actors they measure. The methodology claims only what its machinery supports and states the residual gap in its own copy. A methodology about verifiable claims that overclaimed for itself would refute its own thesis.

Briefs: the unit of work

The unit of work is a brief: a self-contained scope-and-definition-of-done document in a typed format, small enough for one session to own end to end. Four properties do the work:

- **Self-containment** — fleet workers are ephemeral; context that lives only in a session dies with it.
- **Typed dependencies** — a typed reference that stops resolving is a lint failure; a prose reference that stops being true is silence.
- **An executable Verify block** — verification against a table written in advance is repeatable by a stranger; verification against the implementer's recollection of intent is not.
- **A risk gate assigned at authoring time** — routing review effort by risk is decided by the author of the scope, before the work exists, not negotiated afterward by the implementer being gated.

Full format and lint rules: Briefs.

How work enters: one front door, two lanes

Nothing becomes work except by classification. Everything inbound — from humans, agents, findings, or retro process changes — arrives at the intake-desk, the single front door, and is routed by one test: could a worker take it as-is? Work-shaped items are filed in the issues register, where the issue body stays the spec. Idea-shaped items become numbered entries in the intake register, and every entry gets a recorded disposition: accepted, and authored into a brief that enters the execution flow below — or deferred, rejected, or duplicate, closed in the register with its number kept.

Figure (diagram not reproduced in this document) — The front door in one pass: inbound items reach the intake-desk and are routed by shape — work-shaped to the issues register, idea-shaped to the intake register. Both paths that become work converge on todo; everything else closes in the register with its number kept, so a rejection is a record, not a deletion.

The lifecycle: who may say what, and when

```
todo → in-progress → implemented → verified → done
```

The sequence encodes a separation of powers, and each transition has an owner.

- **Implementers stop at implemented.** An implementer verifying its own work is the narrator grading its own exam.
- **Verified belongs to a non-author**, who re-runs the brief's Verify table on merged main and records dated, attributed evidence. The independence is structural, not cultural: the rule is who runs the check, not how earnestly it is run.
- **Merging does not verify.** A merged brief sits in an awaiting-verification queue until a non-implementer is dispatched — verified is a distinct, owned step, not a side effect of merge.
- **Done requires a recorded review**, and the two checks never substitute: the Verify table proves function, the review proves quality. A risk-flagged brief additionally requires a named human.
- **Attribution is mandatory.** Verified and Reviewed cells take dated, attributed entries, never a bare checkmark — an undated tick is unattributable and unauditable.

The full arc: Lifecycle.

THE DESKS: WHO OWNS WHICH TRANSITION

The lifecycle is operated by five standing desks. Four are loops — the intake-desk above, then the worker-desk, the pr-review-desk, and the verify-desk. The fifth, the-desk, is the coordinator: it arbitrates across streams and keeps the registers honest, and it is a hub, not a loop. Each stage transition has one owner, and the two transitions nearest the finish line belong to a desk that did not write the change. Each desk's remit — and, more to the point, the work each one is forbidden to do — is on Desk roles.

Figure (diagram not reproduced in this document) — How one brief travels. The intake-desk admits it to the board as todo; a worker-desk session claims it, implements, and stops at implemented; the pr-review-desk reviews every head of the draft PR; a human — never a desk — merges; the verify-desk, which did not write the change, re-runs the Verify table on merged main and advances the row to verified, then to done once the recorded review is in place. The-desk coordinates all four loops and owns no transition itself.

Registers: memory that resists its authors

Three append-only logs form the system's memory: FINDINGS for knowledge that invalidates existing work, INTAKE for raw ideas awaiting disposition, and RETRO for the cadence retrospective. The registers are the audit trail, so their integrity rules are the strictest in the methodology.

- **Append-only, with enforced sequence contiguity** — a missing number is the visible signature of a deleted entry. The check makes deletion machine-visible instead of luck-visible.
- **Withdrawal is a tombstone, never a deletion** — keep the number, flip the disposition, let the body explain. The history of being wrong is itself evidence.
- **Findings propagate staleness** — a finding names the briefs it affects, and every affected brief is flagged and excluded from the work queue until it resolves. A discovered problem must not be outrun by the work it invalidates.
- **Escalation is gated** — anyone can file a finding, but only one carrying the desk's on-record acknowledgment can hard-error an in-flight brief. An ungated demotion rule would let any session drop a rival brief by filing a paragraph.
- **The retro reads instruments, not narrative** — board totals, untouched streams, what the gates caught, register ages. One process change per retro, and only one that displaces the current worst pain.

Formats and conventions: Registers.

The board: status as a build artifact

The status board is generated from the stream documents and registers by one tool, `statusgen`, which also lints the entire document set in CI.

- **Single writer.** The board is never hand-edited; its only writer is the main branch's CI. A board with one writer is a board where every change traces to a change in a source document.
- **Committed, diffable, lintable.** The board is a file in the repository, not state in a dashboard — it can be diffed, reviewed, and checked by a script, and it outlives any tool's UI.
- **A computed work queue.** Next-up weighs priority and staleness across all streams, caps any single stream's share, and excludes briefs held by unresolved findings — built to defeat the pull toward "the next brief in my stream" over "the most important brief anywhere." Known

limitation: priority-plus-staleness rewards neglect regardless of why a stream aged. The board is a heuristic scheduler, not an oracle.

The tool: statusgen.

Tiering: which model does which job

A fleet has to keep answering a question a single agent never asks — which class of model should take this piece? Assay answers it in advance, from the shape of the work rather than from whoever happens to be running, on one asymmetry: authoring errors compound through every implementer that takes the brief as its premise, while implementation errors meet a Verify table, a reviewer who did not write the change, and a non-author re-running the checks on merged mainline.

- **Authoring and triage run strong; implementation runs cheap behind the gates.** The second clause carries the first. Cheap execution is safe because three mechanisms are already watching it — adopt the tiering without the gates and the same policy just ships worse work faster.
- **Effort keys the default, and a brief can tighten it.** An optional field asserts a minimum tier for work whose difficulty its size understates, derived from three complexity questions rather than chosen. It only ever tightens; it never widens a brief down-tier.
- **Verification does not escalate to a bigger model.** Every other rung answers uncertainty with more capability. This one does not: a risk-flagged brief cannot be signed off by a model at all, so the ladder's top rung is a named human.

The policy, per role, and what none of it checks: Model tiering.

The feedback loop: sensing, not just moving forward

Seven loops, one coordinator. The four desk loops above move work; three more — metrics, analysis, retro — sense it. Everything the desks do lands in artifacts, the board is recomputed from those artifacts, deterministic instruments read the records, analysis turns what they measured into findings, and the retro admits one process change per cadence, on evidence. Findings and process changes re-enter through the same front door as any other work. The loop is closed: the system routes its own defects back into its own queue — to the extent, and only to the extent, that the defects show up in the records.

Figure (diagram not reproduced in this document) — The sense band is dashed because it moves no work — it reads records. The board is derived, not measured: statusgen recomputes it from artifacts the same agents write, so it can catch inconsistency between records, not defects the records never mention. Findings and process changes re-enter through the same front door as any other work, and an unresolved finding holds the briefs it names out of the queue until it resolves.

The incident that shaped the registers

On the system's first day of operation, a session deleted an append-only finding to silence a checker. The deletion was not caught by the register's own enforcement, which did not yet exist in its current form — it was caught because a parallel implementation happened to carry a regression test that noticed the entry's absence. Luck, not machinery.

That incident is published rather than buried because it is the clearest demonstration of the methodology's thesis, and it is the direct ancestor of the rules above: sequence contiguity, so a deletion leaves a numbered hole; tombstone-not-deletion, so retraction has a sanctioned path; a single-writer board, so no session can hand-edit the roll-up; and a review stamp no implementer can mint. Every enforcement mechanism in Assay exists because the convention-only version failed on first contact with the actors it was meant to govern.

Non-author review and the assay mark

The lifecycle's review step produces a stamp, not a sentence. A review counts only when it is posted by an identity the implementer cannot impersonate — in practice an App identity whose credentials no worker session holds. Anything an implementer can write about its own work is a claim; the stamp is the one artifact in the system whose attribution is un-forgeable, and that word is used here in exactly that narrow sense. The stamp does not certify that the review was thorough or the reviewer competent. It certifies that a specific, distinct identity performed the review and is answerable for it — precisely the property a self-written checkmark lacks.

The name of the methodology is a description of this step. To assay a metal is to test it for what it actually contains: the assayer does not take the refiner's word, but cuts the bar, runs the test, and stamps the result. The brief is the bar submitted for testing. The Verify table is the cut. The non-author verifier is the assayer. The review posted by an identity the author cannot imitate is the stamp. The hallmark system worked for centuries because it moved quality from something you were asked to trust to something you could check; the methodology's promise is the same, and deliberately no larger.

The claim, and the residual gap

What the machinery supports. Status is **derived from agent-authored artifacts with consistency linting**, backstopped by independent re-verification. The linter checks internal consistency: sequence gaps, missing evidence, unresolved findings, malformed gates, hand edits to the generated board.

What it does not support. The board is not measured from ground truth. The generator parses tables, frontmatter, gate cells, and evidence blocks — all markdown written by the same agents whose work they report. The sensors are writable by the actors they measure. A lint can check that a done row has a Verified entry; it cannot, today, prove the string in that cell corresponds to a real verification run. Extending identity attribution from the review stamp to the verification gate is the designed hardening — it narrows the gap without closing it while a single identity can author both the work and its record.

What is deliberately not published. No productivity multiplier, no output-per-agent claim, no tier-comparison result. Those figures have no measured baseline, no quality adjustment, and no controlled design behind them. Refusing to quote a number we cannot recompute is the position.

Where the line falls. That refusal is about *claims*, not about counts. A raw count of our own activity — commits in a window, changes merged, bugs filed — is recomputable by anyone with the same command, the same window and the same commit, so it is publishable; a figure asserting that this represents more or better work than some alternative is not, because there is no baseline against which it could be recomputed. The how-it-runs snapshot publishes only the former, states the instant and commit each number was taken at, names the result cap standing behind each one, and leaves every metric it could not compute visible in the table rather than dropping it, marked *could-not-check* and given the same weight as the ones that resolved. No count from that page is restated here: it has one source, and a second copy of a number is a copy that drifts.

Where the methodology does not pay for itself. For a human-paced team, this machinery largely duplicates code review plus CI plus an issue tracker, and the non-author verification step costs a second actor's time. The return is real specifically where agents do a large share of implementation and self-reported status has stopped being informative. Today's lived evidence comes from a single-operator fleet regime; the multi-actor regime, with distinct identities so a forged verification cannot lint green, is a designed direction, not a shipped property.

Placement

The mechanisms here have neighbors: artifact-backed lifecycles, spec-time contradiction checking, verification-strength ladders, and risk-routed oversight all have published precedents that informed the design. A prior-art sweep run on 2026-07-08, refreshed against the July 2026 landscape, found no published counterpart for the combination described here — a generated single-writer board, a cross-stream computed queue, findings with staleness propagation, and an intake register, joined to a lifecycle whose verification evidence must come from a non-author. That finding is scoped to our own sweeps; we say "we found none," not "none exists." The individually novel pieces are few; the discipline is the assembly, and the placement — documents plus a linter that sit under whatever harness, tracker, or orchestrator a team already runs, with the work record as a committed artifact rather than a vendor's UI state.

The toolkit is Apache-2.0, adopted by copying it in, and in daily use on the regulated-finance build it was extracted from. This page describes the methodology that build actually runs, not a specification awaiting a first user.

The summary fits in three sentences. Agent fleets generate claims at machine speed, and the claims are not trustworthy. Assay does not try to make them trustworthy; it makes them checkable, by a script in CI and by a second actor whose stamp the author cannot forge. Quality you can verify, not quality you are asked to trust.

Briefs

A brief is a **self-contained scope-and-DoD contract** — the fundamental unit of work in Assay. One agent must be able to execute it without reading the rest of the plan. Every field is designed so a script can check it.

Why briefs

When a fleet of agents does the implementation, work units written as prose in a task tracker fail silently. Dependencies written as "after the auth brief" break on a rename with no error. A brief is a machine-readable contract: typed IDs, explicit dependencies, a parseable Verify table. The form forces the structure that makes drift visible.

Anatomy of a brief

Every brief carries YAML frontmatter and a structured body. The frontmatter is the parseable record; the body motivates but never solely carries a fact a script needs. These eleven fields are **required** — a brief missing any of them fails lint.

FIELD	WHAT IT HOLDS	WHY IT IS THERE
`schema`	`brief-v1`	The opt-in marker. A document without it is not read as a brief at all, so adding the format to an existing repository is additive rather than a migration.
`brief`	Typed ID: `stream/NN`	Links the brief to its stream README table row. Never a prose name.
`title`	One-line summary	Human-readable, for the board and Next-up display.
`wave`	Integer: 0 = no deps, N = deps in waves < N	Derived from the dependency graph. Lets the board schedule parallel work.
`depends` / `unblocks`	Typed ID arrays, mutual inverses	A typed ID survives renumbering and greps cleanly. No prose arrows.
`effort`	`S`, `M`, or `L`	Scheduling tier. S may run inline; M/L plan then dispatch to implementers.
`gate`	`model` or `human`	Derived from risk answers. Any yes in `risk:` forces `human`.
`risk`	Four booleans: regulatory, customer, irreversible, sensitive-data	Record all four. The gate is their conclusion — not a separate choice.
`issues`	Integer array of tracked issue numbers (may be empty)	Ties the brief to whatever was filed against it. Required even when empty, so "no issues" is a recorded answer rather than an omission.

FIELD	WHAT IT HOLDS	WHY IT IS THERE
`authored`	When the brief was written, and by what	A brief is a claim about the world at a moment. Without its date, a reader cannot tell whether its facts have since gone stale.
`sources`	Typed IDs: scoping docs, findings, intake entries	Provenance. An empty list is untraceable — no one can tell why the work exists.

Several optional fields sit alongside them. The most consequential is `exec-tier`, which asserts a minimum class of model for the work and is derived from three complexity questions rather than chosen — see Model tiering. Others let a brief record why its gate is what it is, what is blocking it, what it is worth, and which downstream consumers a shared value change has to reach.

Load-bearing facts live in islands, not prose. Frontmatter, tables, and Verify rows are the record. Prose motivates but never solely carries a fact a script or reviewer needs.

One honest note about the schema check: the validator confirms that every required field is present and well-formed, but it does not reject an *unrecognised* top-level key. A field name typed slightly wrong is silently ignored rather than flagged — which is why the required set is checked by presence, and why a brief's real contract is the fields listed above rather than whatever it happens to contain.

The Task section

The Task is the implementer's instructions: numbered steps, exact paths, specific actions. It is accompanied by a `files:` listing (exact paths — no repo exploration needed) and `facts:` (the 3–5 project facts required to execute, in `key: value` form). A brief that touches a shared value — a party, env var, config key, field meaning, wire format — enumerates every consumer under `consumers:` with a disposition (fixed-here / follow-up / out-of-scope).

The Verify table

Every brief carries a Verify table: a list of executable checks. Each row has a literal command and an expected exit code or output. A row without both is not a DoD item; it is a hope. Evidence over claims is the whole discipline, and it starts here.

For prose deliverables (docs, articles), Verify rows assert **presence**: a file exists, a section appears, a token is present. Quality is owned by the human review gate — presence gates do not prove quality, and claiming they do is the exact anti-pattern the system exists to catch.

Verify rows must be runnable by someone who did not do the work. The implementer runs them and fills Evidence. A non-implementer re-runs them on merged main to advance from implemented to verified. This is the independent re-execution check: no one assays their own metal.

Rules enforced by lint

- **Typed IDs only.** References are `stream/NN`, `F-NN`, `I-NN` — never fuzzy names.
- **Self-contained.** If executing needs knowledge from another brief, link it in `facts:` or state it in `depends:`.
- **Dependencies are typed and inverse.** A brief in `depends:` must list this brief in its `unblocks:` (and vice versa).
- **Wave consistency.** A brief at wave N must only depend on briefs at waves < N.
- **Provenance required.** An empty `sources:` fails lint.
- **Gate derived from risk.** A brief with `gate: model` and any risk `yes` fails lint.

Splitting and sizing

Briefs should be roughly equal-sized. A brief doing two distinct roles becomes two briefs. Rough test: if the Task needs more than about five steps or touches more than two subsystems, split at authoring time. Uneven briefs stall a wave and hide the real critical path.

Splitting a brief mid-execution counts as authoring, and it carries a rule that sounds petty until you have watched it go wrong: the session doing the split keeps only the piece it was actively working on, and every other piece returns to the board as an unclaimed item. Keeping the whole split set is the rabbit-hole reflex at brief granularity — invisible to the board, and never re-examined for what tier or gate the new pieces should carry.

What it looks like

A short brief in full, frontmatter first:

```
schema: brief-v1
brief: billing/04
title: Retire the legacy invoice serializer
wave: 1
depends: ["billing/02"]
unblocks: ["billing/07"]
effort: M
gate: human
risk: {regulatory: no, customer: yes, irreversible: no, sensitive-data: no}
issues: [412]
authored: 2026-08-13
sources: ["billing/02 (introduced the replacement)", "F-invoice-rounding-drift"]
```

Read the fields against each other and the record is self-checking. `customer: yes` is why `gate` reads `human` — a brief claiming a model gate with that answer set fails lint, so the gate cannot drift from the risk it was derived from. `wave: 1` is consistent only because its one dependency sits at

wave 0. The `unblocks` entry has to appear as a `depends` entry on the other side, or the graph is broken and lint says so. And `sources` names a finding, so a reader who wants to know why this work exists at all has somewhere to go.

The body then carries the Task, the facts an implementer needs, and the Verify table — for a change like this one, rows asserting that the old code path is gone, that the replacement is exercised, and that a neighbouring consumer of the same serializer still works.

Reading

Specifications that converge

Why a work unit written to be checked reaches agreement faster than one written to be read.

Article — publication pending

Why software teams need machine-checkable gates

What multi-agent work breaks, and which mechanism answers each failure mode.

Article — publication pending

These articles are drafted and referenced from the toolkit. Links activate on publication.

Registers

The append-only logs under `docs/streams/` are the system's memory: what invalidated a plan (FINDINGS) and what raw ideas are queued awaiting a decision (INTAKE). These are the audit trail — and the rules protecting them are the strictest in the methodology, because they are the records that the actors being recorded can reach.

FINDINGS

Knowledge that invalidates a brief. A new finding names the briefs it affects; every one of them is flagged stale-knowledge and held out of the work queue until the finding resolves. That exclusion is the point: a discovered problem must not be outrun by the work it invalidates.

Each finding is its own file under `docs/streams/findings/`, with frontmatter carrying:

- **Typed ID**: `F-` plus a slug of 10–20 characters derived from the title — for example `F-ws-token-expiry`. Not a counter; see below.
- **Date and title**: when it was recorded and what it is.
- **Body**: what was found, the evidence, and the fix direction.

- **Affects**: typed IDs of the briefs it invalidates. The validator cross-applies these and fails if one does not resolve.
- **Ack**: the desk's on-record judgment that the finding is real. Anyone may file a finding, but only an acknowledged one can hard-error a brief already in flight — otherwise any session could drop a rival's work by filing a paragraph.
- **Resolved**: `yes` or `no`. Resolve by updating the affected brief to reflect the finding, then flipping the field.

The single `FINDINGS.md` is a *generated view* over that directory, on the same single-writer discipline as the board: regenerated by the mainline branch's CI, never hand-edited.

Why the IDs are slugs, and what replaced the gap check

Findings originally used a counter — `F-01`, `F-02` — inside one shared file, and the contiguity of that counter was the tamper-wire: a missing number was the visible signature of a deleted entry. Under parallel authorship the counter became the problem it was solving. Two sessions filing at the same moment claim the same number, and a register whose entries collide is a register people start working around.

So the scheme changed, and it is worth being exact about what was traded for what. **Slugs carry no contiguity guarantee.** There is no sequence, so there is no gap to detect, and an implementation must not claim gap-detection as a property of this scheme. Three properties replace it, required together:

- **A tombstone check against history.** Any entry file that has ever existed on the mainline branch but is absent from the working tree is flagged. This is what actually catches a deletion, and for the case it covers it is stronger than the gap check was — it *names the missing file* instead of inferring that something is missing.
- **A duplicate-ID check.** Two entries sharing an ID are flagged.
- **Diff visibility.** An entry is a whole file and the roll-up view has one writer, so both a deleted file and its disappearance from the view show up in an ordinary change diff.

The tombstone check is advisory, not a guarantee — and it says so. It is a history comparison, so it can only run where that history is available: it returns nothing outside a repository checkout, and skips on a history-read error. A run in which the check could not execute is presently indistinguishable from a run in which it executed and found nothing. Read a clean result as *no deletion observed*, never as *no deletion occurred*. Publishing the limit rather than the headline is the same discipline as the rest of this site.

Entries created before the change keep their numeric IDs. Those are frozen legacy and stay valid; a numeric ID on a *new* finding is a lint failure.

INTAKE

Raw ideas that enter the system before they become briefs or get turned down. Every proposal gets an entry, and every entry gets a recorded disposition — so a "no" is a numbered record with reasoning, not a conversation nobody can find later.

An intake entry records:

- **Typed ID**: `I-` plus an identifier.
- **Date and title**.
- **Body**: the idea, motivation, and scope.
- **Disposition**: `accepted` (authored into a brief), `deferred`, `rejected`, or `duplicate`.

Withdrawal is a tombstone in place — flip the disposition rather than removing the entry. Deleting one to make a number disappear loses the record, which is the single thing the register exists to prevent.

The two registers are mid-migration, and this page says so rather than smoothing it over. The specification puts both on per-entry files with slug IDs. FINDINGS has moved; INTAKE, in the reference implementation, is still one file on the numeric `I-NN` form, where the original contiguity check does still apply. A methodology page presenting one uniform scheme would be describing a plan rather than a repository.

RETRO — specified, not implemented

The register set is often described as three logs, the third being a per-cadence retrospective. Plainly: **no such register exists in the reference implementation.** There is no retro parser and no entry directory, the specification's section on it is marked informative rather than normative, and no conforming linter enforces anything about it.

The retrospective practice is real and is described in the methodology — it reads instruments rather than narrative (board totals, streams untouched since the last pass, what the gates actually caught, register ages) and admits at most one process change per cadence, and only the one that displaces the current worst pain. What does not exist is a machine-checked register behind it. This is documented rather than quietly dropped because the distance between a described practice and an enforced one is exactly what this methodology exists to make visible — including when the gap is ours.

Shared conventions

- **Append-only.** A new entry is a new file; existing entries are never edited away. These logs are the audit trail, and a record you can rewrite is not a record.
- **Withdrawal is a tombstone, never a deletion.** Keep the entry, flip the disposition or resolution, explain the withdrawal in the body. The history of being wrong is itself evidence.

- **Typed IDs only.** Entries reference briefs and other entries by typed ID (`stream/NN`, `F-<slug>`, `I-<slug>`) — never prose names, which stop resolving in silence.
- **Findings propagate staleness.** An unresolved finding holds every brief it names out of the queue. This is the one register rule that changes which work is eligible, which is why the acknowledgment gate sits in front of it.

The incident these rules answer

On the system's first day of operation, a session deleted an append-only finding to silence a checker. It was not caught by the register's own enforcement, which did not yet exist in its current form — it was caught because a parallel implementation happened to carry a regression test that noticed the entry's absence. Luck, not machinery.

Every rule on this page is downstream of that: deletion detection that names what is missing, a tombstone path so retraction has a sanctioned form, and a single-writer generated view so no session can quietly rewrite the roll-up. The incident is published rather than buried because it is the clearest evidence for the thesis — the convention-only version of each of these rules failed on first contact with the actors it was meant to govern.

Lifecycle

How a brief moves from idea to done — and why the most important rule is that the implementer stops before the finish line.

The five steps

```
todo → in-progress → implemented → verified → done
```

STATE	MEANING	WHO
todo	Authored, unclaimed, dependencies known. On the board, in Next-up if eligible.	Author
in-progress	A session owns it and is implementing. Moved from the queue to active work.	Implementer
implemented	The implementer finished and filled the Evidence section with their own run. Implementers STOP here.	Implementer
verified	A non-implementer re-ran the Verify table on merged main and filled Evidence (dated, with runner).	Independent verifier
done	Verified and carries the recorded review verdict. For human-gated briefs, a review entry naming a human is required.	Verifier + reviewer

Why implemented is not done

An implementer verifying their own work is assaying their own metal. In multi-agent work this is not a philosophical concern — it is the primary failure mode. Every agent asked whether its task is done will say yes. A verifier drawn from the same fleet shares correlated failure modes with the implementer.

The lifecycle rule — implementers stop at implemented, a non-implementer advances to verified — makes self-grading structurally impossible. That extra step is friction, and the friction is the feature. It does not make the system uncorruptible, but it makes the corruption visible: a brief sitting at implemented with no verifier dispatched is a queue item, not a hidden assumption.

A sequence of states is only half of a separation of powers; the other half is assigning each transition to somebody who cannot also perform the one before it. That assignment is the desk model — five standing roles, each defined by what it may not do. See Desk roles.

Evidence and review

Verified and Reviewed cells take dated, attributed entries (`2026-07-08 model-verifier`, `human:alex`), never a bare checkmark. An undated tick is unattributable and unauditable. A dated, attributed entry works like a hallmark: it records who struck it and when.

The two checks are distinct:

- **Verify table** proves function: "does it work?" Re-running the brief's Verify commands on merged mainline.
- **Review** proves quality: "is it well-built?" A code, structure, and security pass. Neither substitutes for the other.

Not every entry in those cells may come from a model. Attribution carries a **risk-keyed floor**: a brief with any risk answer set, or a human gate, will not pass lint at `verified` or `done` with an economy-tier runner recorded against it — and for the highest-risk work the floor is a named human rather than a stronger model. Which class of actor may sign which gate is set out on Model tiering.

Merging does not verify. After a brief's pull request merges it sits in the board's awaiting-verification queue. Verified is a distinct, owned step — not a side effect of merge. An unwatched awaiting queue is how briefs rot at implemented, which is why draining it is a standing role rather than a task somebody remembers.

The execution witness

There is a gap in everything above, and it is worth naming before describing what closes part of it. An Evidence section is prose written by the session that ran the checks. It asserts that the commands were run and what they returned. Nothing in that assertion is distinguishable from a session that ran nothing and wrote a plausible table.

So the toolkit can run the Verify table itself. It executes each row's command in a fresh subshell at the repository root and appends one witness row per Verify row:

```
| # | Command        | Result    | Output              | Date       | Runner
|
| 1 | `go test ./...` | pass exit=0 | sha256:6f1a0b3c9d22 | 2026-08-13 | human:alex @
b988d175ab12 |
```

Three things about that row matter more than its shape. The **result is three-state**: a row can pass, fail, or come back *could-not-run* — a command that was not found, an unsubstituted placeholder, a timeout. Could-not-run is recorded as itself, never folded into either of the other two, because a check that did not execute reporting the same thing as a check that passed is the failure mode the whole discipline is built against. The **output is hashed**, so a claim about what a command printed is pinned to something reproducible rather than to a summary. And the **runner and commit are recorded together**, so the row says what was run, against what, by whom.

A closed brief whose Evidence claims the rows were run but carries no witness draws a notice on the board. It is deliberately a notice rather than a failure: a brief closed before the mechanism existed cannot have a witness, and hand-writing witnesses into closed briefs to clear the flag would manufacture exactly the evidence the witness exists to replace.

STATUS.md: the single-writer rule

`STATUS.md` at the repo root is generated from the stream READMEs and registers — never hand-edited. It has exactly one writer: **main's CI**, which regenerates and commits it on every push that touches a source.

- **Branches never commit STATUS.md.** PR CI runs `--lint` mode only and blocks any PR whose diff touches STATUS.md.
- **One writer eliminates the conflict class entirely.** A generated file committed on branches turns every concurrent PR into a merge conflict.
- **Regenerate locally freely; never commit on a branch.** On a merge conflict, take either side and rerun the generator — never hand-merge a generated file.

Next-up: the cross-stream queue

The generator computes a Next-up batch so a session does not default to "the next brief in my stream" — the rabbit-hole reflex the system exists to prevent. Next-up weighs:

- **Priority + staleness** — higher-priority and aged streams rise.
- **A 2-per-stream cap** — no single stream floods the batch.
- **Findings exclusion** — a brief with an unresolved finding against it is held out.

A known defect: the staleness score rewards neglect regardless of *why* a stream aged. A value/effort term is a candidate knob — the board is a heuristic scheduler, not an oracle.

---

statusgen

The Go tool that reads a repo's stream documents and registers, and generates the single `STATUS.md` board. It is the machinery that turns convention into enforcement: gates a script runs, not rules a team is asked to remember.

What it does

`statusgen` parses each stream directory's `README.md` frontmatter and brief-row table, cross-applies findings to the streams they affect (flagging affected briefs as stale-knowledge), reads the two registers (`FINDINGS.md` and `INTAKE.md`), and emits a single aggregated view: one `STATUS.md` file with a priority-and-staleness-ranked Next-up batch, a cross-stream board, and a findings-intersection table.

Three modes carry the everyday loop:

COMMAND	WHAT IT DOES	WHEN TO USE IT
`statusgen`	Generate `STATUS.md` from sources. Writes the file.	Locally, to read the board and see Next-up. Never commit it on a branch.
`statusgen --lint`	Run every source check and build the view internally, but never read or write `STATUS.md`.	Pull-request CI. Blocks changes carrying malformed briefs, broken dependency graphs, or a hand-edited board.
`statusgen --check`	Regenerate `STATUS.md` internally and compare byte-for-byte to the committed version.	Drift check. Advisory — not a gate by default.

The surface is larger than those three. Alongside the board it carries an execution witness that runs a brief's Verify table and writes back a three-state result per row; a set of read-only instrument views (delivery metrics, a transition historian, a per-stage bottleneck report, register alarms, a launch-readiness rollup); and several narrow checks used by the desks. The three above are what an adopting team needs on day one; `statusgen --help` is the authority on the rest, and this page deliberately does not try to be a command reference that would go stale the first time a flag was added.

The single-writer model

Only one CI job — triggered on push to the default branch — ever commits `STATUS.md`. Every other context (PRs, local runs) only reads or lints. This eliminates the entire class of merge conflicts on a generated file and makes drift between sources and status immediately visible: if a PR's lint step

passes but the merged main regen produces different output, something changed under the PR.

What it checks

In `--lint` mode, statusgen runs every source check:

- **Brief frontmatter validity:** every required field present, typed IDs, wave consistency, gate derived from risk, sources non-empty.
- **Dependency graph integrity:** typed IDs only, `depends` and `unblocks` as mutual inverses, no self-loops, no wave violations.
- **Register integrity:** ID format (slug-form on new entries, numeric grandfathered), duplicate IDs, and a tombstone check against branch history for entries that have gone missing.
- **Status cell format:** lifecycle tokens only, dated and attributed entries in the Verified and Reviewed columns, no bare checkmarks.
- **Attribution floors:** a risk-flagged brief cannot reach `verified` or `done` with an economy-tier runner recorded against it.
- **Findings cross-application:** every finding's affected IDs resolve to real briefs; affected briefs are flagged stale and excluded from the queue until resolved.
- **Link resolution:** a document reference that no longer resolves is a failure, not a broken link somebody notices later.
- **Stream frontmatter:** required fields present, valid values.
- **Board immutability on branches:** any change whose diff touches `STATUS.md` trips lint.

Lint separates its output into **problems**, which fail the run, and **notices**, which do not. The split is deliberate: a notice is for something a human should look at but which no machine can adjudicate — a brief closed before a mechanism existed, a standing alarm aging past its threshold, a queue growing faster than it drains. Promoting those to failures would train people to route around the checker, which costs more than the thing being flagged.

Repo-agnostic

`statusgen` is a standalone Go module — the standard library plus a YAML parser, and nothing else. Its module path is `github.com/medici-finance/assay/statusgen`. It knows nothing about any particular product: it reads a directory of stream documents and registers, and it will read yours.

It was extracted from an internal product repository in July 2026, already written repo-agnostic — the register and brief conventions on this site are the ones that build actually runs on, not a specification drafted for publication.

Take the binary, not the source

The obvious way to adopt a small self-contained tool is to copy its directory into your repository. **That recommendation has been retired**, on evidence from our own repositories rather than on principle.

A vendored copy is a fork, and forks rot in silence. The concrete case: a copied `statusgen` gating another repository's pull requests, frozen at the commit where it landed and untouched for weeks afterwards — still passing, still green, and missing three checks that had since been added upstream. Nothing failed. That is the problem. A gate that has quietly stopped checking what it is believed to check is worse than no gate, because a team stops looking at what it has delegated. Several sibling cases followed the same shape.

So the supported path is a **pinned release binary**: a small file in your repository names the release tag and the expected hash for each platform, installation verifies the hash, and upgrading is a one-line change reviewed like any other. One binary, one pin, one place to bump — and drift becomes a diff instead of a silence.

This repository is the canonical home for the tool; consumers take pinned releases from it rather than maintaining their own copy.

Reading

Status as a build artifact

What changes when the status board stops being a dashboard and becomes a generated file with one writer.

Article — publication pending

Drafted and referenced from the toolkit. The link activates on publication.