# SEAHORSE v1.0 — What Actually Makes an Agent Code Well: Concurrency, a Bible, Four Premises, and the Harness

Oscar Serra · Independent Research

5 August 2026


## Abstract

An agent that writes code well is not primarily an agent that writes good code. It is an agent that can (a) work alongside other agents on one tree without corrupting it, (b) find out what is already true of the system before acting, (c) hold a small number of beliefs that survive contact with a large codebase, and (d) run on a harness that gives it the primitives for the first three.

This paper takes those four in order. §1 describes ORCA, a lease-based orchestrator for concurrent multi-agent editing, and argues that the correct unit of mutual exclusion is the *file during application*, not the task. §2 describes a structured design corpus — a "bible" of single-owner optics with executable verification blocks — and gives an honest accounting of what it did and did not prevent, including four features that were simultaneously dead while 339 documentation invariants passed. §3 states four premises about belief that separate an agent which makes progress from one which stalls, each grounded in a specific incident where the belief was violated and the cost was measured in months. §4 surveys harness capabilities.

The paper's central negative result is in §2, and it is the reason §3 exists: **a documentation gate asserts shape, not liveness.** Every invariant in the corpus can pass while the system it describes does nothing at all. On 2026-08-04 that was not hypothetical — four headline features were dead simultaneously, one of them for eight weeks, with a fully green gate.


**Provenance note.** This paper is written from one system's operating record, not from a survey. Every incident it cites is first-party, dated, and reproducible from the reference implementation's git history, its bug log, and its ledgers. Where the wider literature has measured something relevant it is cited; where it has not, the paper says so rather than borrowing a number. The reference implementation is a fork of an open-source agent harness, operated continuously by a single architect with several agents working the same tree at once.

**A word on the codename.** SEAHORSE is the first J-series name taken from the sea rather than from neuroanatomy, joining the tooling it describes — ORCA, the parallel coding orchestrator, and MERMAID, the diagram language the design corpus renders with. The collision is worth naming: the hippocampus is named after the seahorse (Greek *hippokampos*), and J2 already holds HIPPOCAMPUS. So this paper and the memory paper are, etymologically, the same animal. That is an accident, but not an unpleasant one: this paper is about what a coding agent must remember in order to work.

# 1. ORCA — concurrent agents on one tree

## 1.1 The problem is narrower than it looks

The obvious framing of multi-agent coding is "how do several agents divide a task?" That framing produces schedulers, planners and role hierarchies. It is the wrong first question, because the thing that actually breaks is much more specific: **two agents writing the same file at overlapping times.** Everything else — reading, reasoning, drafting, testing — is safely parallel.

This matters because the expensive part of an agent's work is the part that is safe. Reading a subsystem and deciding what to change can take minutes and many thousands of tokens. Writing the resulting patch takes milliseconds. A design that serialises agents at the *task* level pays the expensive part serially to protect the cheap part.

## 1.2 The three phases, and why the lease is where it is

ORCA splits each unit of work into three phases with different concurrency rules.

**Phase A — draft, fully parallel, no lease.** Every agent reads the tree and produces an exact patch for its own edit-unit. No agent holds anything. This is where nearly all the wall-clock and nearly all the tokens go, and it is embarrassingly parallel because reading is not a mutation.

**Phase B — apply, serialised per file.** An agent takes a short-lived lease on the files its patch touches, applies, and releases. Units with disjoint file sets run concurrently; units sharing a file serialise. A patch that has gone stale because another unit landed first is **re-derived**, not force-applied — the agent goes back and redrafts against the new state.

**Phase C — commit, one commit per unit, serialised.** Each applied unit becomes its own commit, staging *only that unit's files*. Never `git add -A`.

That last rule looks like fussiness and is not. On this system a second agent's uncommitted work is present in the tree at essentially all times — over the session that produced this paper, a parallel session held between one and five modified files continuously. `git add -A` from either agent would have swept the other's half-finished work into a commit that claims to be about something else. The rule is not hygiene; it is the difference between "concurrent agents" and "one agent and a hazard".

## 1.3 What the lease is actually protecting

It is worth being precise, because it determines the design. The lease does not protect *coherence of intent* — two agents can hold entirely compatible plans and still corrupt a file. It protects a much smaller invariant: **that a patch is applied against the tree state it was computed from.**

This gives the correct failure behaviour for free. If a lease is contended, the loser does not fail, it *waits and re-derives*. Staleness is a normal, expected event with a defined recovery, rather than an error. Systems that treat concurrent edit as an error tend to acquire a retry loop that re-applies the same stale patch, which is how you get a merge conflict rendered as a corrupted file.

## 1.4 What this does not solve, stated plainly

ORCA makes concurrent edits safe. It does not make them *correct*. Two units can each be individually right and jointly wrong — the canonical case being a shared contract where unit A changes a function's signature and unit B adds a caller using the old one. Both apply cleanly; neither is stale; the build breaks.

The mitigation the reference implementation uses is a contract owner: when units are interdependent, one unit owns the shared interface and the others declare a dependency on it, so the dependent units draft *after* the owner has applied. This is a real constraint on decomposition and it means the orchestrator cannot be handed an arbitrary task list. Independence is a property the caller must establish; ORCA enforces safety, not independence.

### 1.5 Relation to the literature

The lease pattern is not novel in itself — it is pessimistic locking at file granularity, and the "draft optimistically, validate on apply" shape is optimistic concurrency control with re-derivation instead of abort. What appears less common in agent frameworks is the *phase split*: applying the two disciplines to different phases of the same unit of work, so the expensive phase gets optimistic concurrency and the cheap phase gets pessimistic locking.

## 2. The bible — a design corpus with executable invariants

### 2.1 What it is

The reference implementation carries a structured design corpus of roughly thirty markdown "optics", each owning one class of fact: component topology, failure propagation, configuration shape, session naming, cron registry, branch policy, and so on. Four properties define it.

**One owner per fact.** Any given fact is stated in exactly one optic; others cross-reference. This is the property that decays first and matters most — the same fact in two files becomes two facts as soon as one is edited.

**An apex document that outranks the rest.** `FOUNDATION.md` holds the principles every optic derives from. When an optic contradicts it, the optic is the bug.

**Executable verification.** Each optic carries `verify:` blocks in its frontmatter — named checks with a one-line command. A single command runs all of them (currently 339). A documentation claim that cannot be checked is prose; one that can be is an invariant.

**Derivation over enumeration.** Anything countable is computed at check time, never written down. An earlier version of the index asserted a file count in prose and `>= 18` in its verify block; both were stale within weeks, and the floor had been outgrown so thoroughly that eleven optics could have been deleted without failing it.

### 2.2 Three different jobs, three different homes

The corpus went through one significant structural correction, and it generalises. A governance principle in the system's own documentation read, in substance: *"if I might forget it, enforce it with code; documentation is for explaining why, not for enforcing rules."* An agent read that as a rule about **how documentation should be written** and began pasting enforcement programs inline into the markdown. The intended meaning was **runtime enforcement of the agent's own behaviour** — it exists because a language model forgets documented instructions, so anything that must be obeyed has to sit on the execution path.

The resolution was not a corrected rule but a *disambiguated* one — naming the three distinct jobs that had been sharing one home:

| Job | Home | Wants |
|---|---|---|
| **EXPLAIN** | prose and diagrams | ambiguity is allowed, often the point |
| **ENFORCE** | running code: hooks, gates, guards | must not depend on being remembered |

| Job | Home | Wants |
|---|---|---|
| **CHECK** | scripts behind a one-line pointer | linting, review, and its own tests |

The failure was putting CHECK inside EXPLAIN. Once the three are named, the original rule is unambiguous and the misreading is unavailable.

> **Ambiguity in a governance document is not a writing defect to be polished away. It is an attack surface, and the mitigation is to name the distinct jobs a rule could be serving.** A one-line principle that reads cleanly to a human is exactly the kind of artifact an agent will over-apply, because natural language does not carry its own scope.

### 2.3 The honest assessment

Oscar asked for a candid answer to "how much better does this let you code?" Here it is, in both directions.

**What it demonstrably bought.**

*Orientation without archaeology.* The single largest cost in a large unfamiliar codebase is establishing what is true before acting. A per-optic corpus with a routing index turns "which subsystem owns this?" from a twenty-minute search into one file read. This is real and it compounds.

*A place to put the reason.* Most code comments record what a line does. The expensive knowledge is why a specific value is 8,000 and not 12,000, and what happened the last time someone raised it. An optic gives that a home with a stable address.

*Regressions that fail the build.* Executable invariants convert a documented rule into a gate. Several checks in the corpus have bitten, and two were repaired during this session's work because the code had legitimately moved past what they asserted — which is the system working.

*A ratchet discipline that survives backlogs.* Several gates enforce a measured status quo that may fall but never rise. This matters because a gate demanding perfection on day one gets switched off by Friday. Collapsing twelve duplicate token estimators is not a session's work; forbidding a thirteenth is free.

**What it demonstrably did not buy, which is the more useful half.**

On 2026-08-04, with all 339 invariants passing, four headline features were simultaneously dead:

| Feature | State |
|---|---|
| Fractal reflection | 2,466 recorded runs, **zero** successes, for eight weeks |
| Semantic memory search | 5,258 chunks indexed, **zero** vectors |
| Overseer nudges | 188 permission refusals, **zero** nudges |
| ENGRAM ingestion | a no-op on **every** turn since 2026-07-28 |

Not one of these was caught by the corpus, and it is worth being exact about why, because the reason is structural rather than a matter of insufficient coverage.

> **A documentation invariant asserts SHAPE, not LIVENESS.** It checks that a file exists, that a symbol is exported, that a call site delegates to the canonical helper, that two numbers agree. Every one of those can be true of a subsystem that has not executed successfully in two months.

The corpus also failed to prevent a command-injection vulnerability in the harness's own enrichment path, described in the companion security paper — it had an optic on failure propagation, an optic on inspection primitives, and a rule about canonical derivations, and none of them asks "does any string reach a shell?"

**The correct conclusion is not that the corpus is worthless.** It is that documentation gates and liveness gates are different instruments measuring different things, and a system with only the first will believe itself healthy while dead. The reference implementation's response was to build the second: a derived capability registry that scores every plugin, hook, RPC and store as OBSERVED, DECLARED or BLIND, and ratchets the blind count downward. Its first measurement found that **24% of the system's capabilities had any signal at all**, and that the entire 185-method gateway RPC surface had none.

That number is the honest measure of what the design corpus was worth on its own: it made the system explicable, and it left three-quarters of it unwatched.

## 2.4 Inherited enforcement: how a check written in March directed work in August

The three-jobs split in §2.2 has a failure mode that only appears over time, and it is worth its own subsection because the reference implementation walked into it while this paper was being written.

In March 2026, contributors to the upstream project this system is forked from added a CI check forbidding an extension from importing core code by relative path. Their history shows it was taken seriously: introduced with a baseline of known violations, then — four days later, in a commit titled *"remove relative extension boundary escapes"* — the violations were cleaned to zero and **the baseline mechanism was deleted**, so the count could never climb again. It was tightened twice more that spring and was still being maintained in July. For that project it is a live, load-bearing constraint, and the reasoning behind it is legible from what they do: they keep ~150 extensions each with its own `package.json` and treat an extension as a detachable unit.

The check carries **no comment explaining any of that.**

In August, in a fork those authors have never seen, whose git lineage is no longer comparable to theirs, which publishes nothing to a package registry and takes no further merges from them, that check reported 39 violations. What happened next is the interesting part, because nobody at upstream caused it and nobody at the fork wanted it:

- The agent adopted the check's vocabulary without inspecting it — writing *"39 violations"* as though violation were a fact about the code rather than a verdict of one program.
- It proposed the remedy the check implies: roughly 26 new SDK entry points and 39 import rewrites, about 65 file edits, producing **zero behavioural change** — every one of those imports resolves correctly today and always has.
- It described a publish as *"blocked"* by them.
- Correcting this took the system's owner asking four questions in plain language: *what does a violation mean, who sets the rule, what rule is it, and toward what end was it designed?* None of those four questions is answerable from the check itself, and the agent had answered none of them before proposing the work.

**The delivery mechanism is an executable that fails.** Not a convention, not a style guide, not a document — those decay into being ignored. A failing check *compels*, it has no expiry date, it names no author, and it states no scope. It is a standing instruction to every future contributor, carried in the repository, that outlives the conditions that justified it and cannot notice when they lapse.

An agent is unusually susceptible to it. A red build is close to the only thing in an agent's environment that resembles an objective fact — everything else is text it must interpret — so it

gets treated as one. Told that something is a violation, the agent's default is to make the light green, not to ask on whose authority the light was installed.

**And note the inversion against the rest of this paper.** §2.3's failures were gates that were *green and meaningless*: 339 invariants passing over four dead features, shape mistaken for liveness. This is the opposite — a gate *red and meaningless*. The red case is the more expensive of the two. A false green costs nothing until something breaks; a false red demands work immediately, keeps demanding it, and the work it demands looks like diligence.

Two practices follow, and the second is the one this system got wrong twice in an afternoon.

**A gate must carry its own rationale and its own scope**, because the single thing a check cannot do is tell you when it stopped applying. Upstream's intent was in fact recoverable — it was legible in four months of their commit history, freely readable, and nobody looked until prompted. An undocumented gate does not preserve a decision; it preserves only the decision's *effect*, stripped of the conditions that made it correct.

**And when you do decide a rule no longer applies, define the exemption from something checkable.** Asked to scope the check to upstream-owned code, the agent wrote a predicate matching directory names beginning `tinkerclaw-`. Three of the five offending extensions — `overseer`, `whatsapp-tinkerclaw`, `auth-reload` — are fork-authored and do not carry that prefix, so the exemption was wrong for the majority of what it was written for. A checkable definition was sitting right there: *does this path exist in* `upstream/main`*?* That is a command, not a guess. Inventing a second definition of an existing concept is the duplicate-derivation failure of §2.1 wearing the clothes of a fix — and it very nearly shipped as a green gate, which is the worst outcome available: a rule that appears enforced, is not, and now lies about which code it covers.

## 3. Four premises about belief

The difference between an agent that makes progress on a large system and one that stalls is not mostly capability. It is a small number of beliefs about what to do when the evidence is incomplete. Each of the four below is stated as an instruction, followed by the incident that proves it.

### 3.1 Never answer "no" before trying

The failure mode is a confident, well-argued impossibility claim that was never tested. It is especially seductive because the argument is usually *locally* sound — it just rests on a premise nobody checked.

**The incident.** A comment in the reference implementation justified writing a prompt to a temporary file and shelling out with `cat` rather than passing it as an argument: *"too large for CLI* `--text` *argument"*. The prompt is 811 bytes. The per-argument limit on the platform is 131,072. The claim was wrong by a factor of 160, had been true of nothing, and had bought a `/bin/bash` on a hot path plus a predictable file in a world-writable directory. It survived for as long as it did because it was phrased as a fact and read as one.

The instruction is not "be optimistic". It is: **an impossibility claim is a hypothesis, and the cost of testing it is almost always lower than the cost of the workaround it justifies.**

### 3.2 Trace the assumptions under a conclusion

When a conclusion is wrong, the error is rarely in the reasoning. It is in a premise that was acquired so early it stopped looking like a premise.

**The incident.** A cognitive lane in the reference implementation timed out on roughly one run in eight. The ceiling was 120 seconds, documented as *"triage runs at low thinking and should finish well under this."* Measured across 291 recorded runs: median 66 seconds, **90th percentile 128.5 seconds**, maximum 260. The ceiling had been set from an assumption about how long the work *ought* to take, and no one had re-derived it from the ledger the system was already writing.

The tell is a specific kind of sentence: *"X should be well under Y."* That is a prediction wearing the clothes of a specification.

### 3.3 If the conclusion says it cannot be coded, revise the assumptions

This is the strongest of the four and the one Oscar states most forcefully: everything can be coded. When analysis terminates in "this is not possible", the correct response is to treat that as evidence of a bad premise rather than a property of the world.

**The incident.** The question "which of this system's capabilities are observable?" looked unanswerable — there is no registry of capabilities, they are spread across plugins, hooks, RPCs, crons, stores and UI panels, and any hand-written list would be stale on arrival. The premise doing the damage was *a capability list must be written.* Dropping it makes the problem easy: **derive the list from the tree on every run.** The result is a 576-row registry computed in under two seconds, with each row carrying the query that produced it so any human can re-derive it by hand.

The pattern generalises: an impossibility usually attaches to one *implementation* of the goal that has been silently promoted to the goal itself.

### 3.4 A verification you did not keep is an anecdote

The weakest link in agent work is not doing the check. It is doing the check once, in a shell that scrolls away, and then writing "verified" in a durable document.

**The incident.** A security fix was recorded in a paper as *"verified by differential test: the identical malicious path executes under the old pattern and does not under the new."* The verification genuinely happened. **No such test existed in the repository.** Nothing prevented the pattern returning, and the sentence would have been read for years as evidence that something guarded it.

There is a sharper form of this rule, learned the same day. When the test was finally committed, it kept its *control* case: it first proves the old pattern really does execute the payload on this machine, and only then that the new one does not.

> **A safety assertion without a control is unfalsifiable.** "The malicious input did nothing" passes equally against a real fix, a dud payload, a missing binary, and a typo in the fixture. Only the control distinguishes "we are safe" from "the test is broken".

### 3.5 Why these four and not a longer list

Each of the four is a rule about *the relationship between a belief and its evidence* — how an impossibility claim relates to a test (3.1), a conclusion to its premises (3.2), a goal to an implementation (3.3), and a documented claim to a retained artifact (3.4). They are not coding advice. An agent that holds them will re-derive most coding advice on demand; an agent that does not will follow coding advice into a wall and describe the wall accurately.

## 4. The harness

## 4.1 What a harness is for

Sections 1–3 are not achievable by a better model. They are achievable by a model with the right primitives underneath it. A harness is the layer that decides which of the following an agent can do at all, and the differences between harnesses are larger than they appear from a feature list.

The primitives that turned out to matter, in the order they mattered:

**Deterministic orchestration over non-deterministic workers.** The ability to write control flow — loops, conditionals, fan-out, barriers — *as code*, and have each step dispatch an agent. This is the primitive §1 rests on. A planner that asks a model "what should we do next?" at every step inherits the model's variance at every step; a script that says `for each unit: draft, then apply under a lease` does not. The valuable property is not parallelism, it is **reproducible topology**: the same script produces the same shape of work every time, and only the contents vary.

**Isolation on demand.** Giving a worker its own git worktree so that parallel mutation cannot collide at all. This is strictly stronger than a lease and strictly more expensive, and the right default is the lease — but the escape hatch has to exist for the cases where two units genuinely cannot share a tree.

**Structured output that is enforced, not requested.** Asking a model for JSON and parsing the result is a retry loop you will write badly. Having the harness force a schema at the tool-call layer, so a mismatch is retried before it reaches your code, removes an entire class of glue.

**Hooks on the execution path.** The ability to run a check *before* a tool call, synchronously, with the power to deny it. §2.2's ENFORCE column is only available if this exists. A harness that can only advise the model — "please do not do X" — cannot implement a guard, because advice is subject to the same forgetting the guard exists to defeat. The reference implementation's deterministic action floor works precisely because it denies the call rather than asking nicely.

**Background work with completion notification.** Long operations that do not block the turn, and that wake the agent when they finish. Without it, an agent polls — and polling loops are where agents burn budget doing nothing.

**Session lifecycle events.** And here is the lesson this session paid for, which belongs in any honest treatment of the primitive rather than in a footnote.

## 4.2 Lifecycle events: prefer the guaranteed one

The reference implementation syncs a Claude Code session's curated memory into the persistent agent's long-term store, so that what one ephemeral session learns is available to the other. It was wired to a `SessionEnd` hook — the semantically obvious choice, since a summary belongs at the end.

It produced nothing for three days. The script was never broken; a dry run three days later emitted 33 pending events immediately. **`SessionEnd` had simply not fired**, because long-lived sessions are far more often abandoned, compacted, or killed than cleanly ended.

The fix is to trigger on an event that is guaranteed rather than one that is tidy. A session that never ends still always *starts*, so the same sync now runs on `SessionStart`, picking up whatever the previous session left behind. Both are wired; a state file dedupes, so whichever fires first wins and the other reports "nothing new".

> **Trigger on the event that is guaranteed to happen, not the one that is semantically correct.** "At the end" is where a summary belongs conceptually, and is the moment least likely to arrive.

There is a second lesson stacked on the first. The script reported only to stderr, and a hook's stderr goes nowhere, so "did it run?" had no answer short of comparing two file mtimes and inferring. It now writes a receipt on *every* outcome — including "nothing new" — because **"ran and had nothing to do" and "never ran at all" being indistinguishable is exactly what hid the outage.** A receipt with a stale timestamp says "not running" as loudly as a missing one; an absent log line says nothing at all.

### 4.3 What is still missing

Stated as gaps rather than complaints, because each is a concrete thing a harness could provide.

**No first-class notion of a capability.** Every harness surveyed can register a tool, a hook, a command. None can answer "what can this system do, and which of those things has run recently?" The reference implementation had to derive that from the tree because nothing else could produce it, and the derivation found that three-quarters of its own capabilities had no signal at all.

**Instrumentation is per-author, so coverage follows scar tissue.** Where instruments exist, they exist because something once hurt. In the reference implementation, hooks were 71% covered and the RPC surface 0% — not by design, but because hooks had broken visibly before and RPCs had not. The fix in both cases was to instrument the *dispatch chokepoint* rather than the individual sites: one wrapper at the hook registrar covered every plugin hook; one counter at the gateway dispatcher covered 337 methods. A harness could offer this natively, and none does.

**The UI is structurally unobservable.** The reference implementation's browser-side interface contains zero instrument declarations, because the liveness registry is a Node-side facility and the UI does not run there. Thirty-one user-facing capabilities are therefore invisible to the only report anybody reads. This is a boundary problem, not neglect, and it is the reason the architect's recurring complaint — *"the visualizer does not work nor spits anything useful"* — was so hard to act on: there was no channel through which it could have reported its own failure.

## 5. What this adds up to

The four parts are one argument.

An agent codes well when it can act concurrently without corrupting shared state (§1), orient itself in an unfamiliar system without archaeology (§2), hold beliefs that fail loudly rather than quietly (§3), and run on a substrate that makes the first three expressible (§4).

The negative results are the load-bearing ones. A design corpus with 339 passing invariants coexisted with four dead features, because **shape is not liveness**. A security fix documented as "verified by differential test" had no such test, because **a verification you did not keep is an anecdote**. A memory bridge wired to the semantically correct lifecycle event produced nothing for three days, because **the tidy event is not the guaranteed one**. A ceiling set from a plausible assumption cut off one run in eight for two months, because **"should be well under" is a prediction wearing the clothes of a specification**.

Each of those is cheap to fix once seen and nearly impossible to see without an instrument. That is the whole thesis: the binding constraint on agent coding quality is not reasoning. It is **observability of the agent's own work**, and it has to be built deliberately, because every default in the stack — a silent early return, a catch-only log, a hook whose stderr is discarded, a status line that says ok — conspires to report health.

## 6. Related work, and where this paper's claims sit against it

This section is deliberately short and deliberately hedged. It reports only what was checked against a primary source during a structured mining pass; where a claim comes from a companion paper's research rather than this one's, it is attributed rather than restated as an independent finding.

## 6.1 Memory and retrieval

The agent-memory field has converged on four mechanisms, and the reference implementation already contains fork-local ports of all of them — which is itself the finding.

**Bi-temporal fact invalidation.** Zep/Graphiti (arXiv:2501.13956; https://github.com/getzep/graphiti) gives every edge four timestamps, separating *when the system learned a fact* from *when the fact was true*, so a contradiction closes a validity interval rather than deleting a row. The reference implementation ports this faithfully. §2's lesson applies exactly: the port is correct and, for the cross-corpus case, structurally unable to fire, because fact identity includes the corpus tag.

**Write reconciliation as a classification.** Mem0 (arXiv:2504.19413; https://github.com/mem0ai/mem0) compares each candidate fact against similar existing memories and emits ADD / UPDATE / DELETE / NOOP, separating extraction from update. Ported here, and wired to an always-add default — present, inert.

**Rank-based fusion.** Graphiti's search recipes fuse parallel retrievers with Reciprocal Rank Fusion, MMR, node-distance or a cross-encoder (https://help.getzep.com/graphiti/working-with-data/searching). RRF fuses on *rank*, which is what makes heterogeneous retrievers combinable at all. The reference implementation instead sums raw scores from BM25 and cosine retrievers plus a recency term and reranks the mixture — arithmetic on incommensurable units, and the clearest single defect the mining pass found.

**Provenance to the source episode.** Graphiti retains lineage from every derived fact to the ingestion that produced it. The reference implementation's event metadata carries seven fields and no origin, which is why a merged corpus could not be evaluated even if it were measured.

The synthesis: **the gap between this system and the field is not algorithmic.** Every headline mechanism is present. What is absent is provenance, measurement, and — in two cases — a wiring decision that leaves a correct implementation switched off.

## 6.2 Ambiguity, restraint, and why the harness must supply them

The companion security paper (J9, notes 8–10) surveys the benchmark landscape for agent *disambiguation* and *restraint*, and the results reported there are directly relevant to §3 of this paper: they suggest that knowing **when not to act** is a separate competency from knowing **how to act**, and that current models are trained for the second. That survey's figures are not reproduced here — they belong to that paper, and restating them second-hand is precisely the error §3.4 warns about.

The structural conclusion this paper draws from it is narrower and stands on its own evidence: **every incident in §2 and §3 was a case of the system acting confidently on an unverified premise**, and in none of them would a better model have helped, because in none of them did the model have access to the disconfirming fact. The ceiling was wrong because nobody read the ledger. The "too large for `--text`" claim was wrong because nobody measured the file. The differential test was missing because nobody looked for it. Restraint supplied by the model would not have found any of these; an instrument would, and did.

### 6.3 Concurrency

§1's lease model is, as stated there, a composition of two textbook disciplines rather than a new one. The mining pass surveyed the major open-source multi-agent frameworks and found that they differ mostly in *role vocabulary* — planner/worker, manager/agent, crew/task — rather than in coordination semantics over a shared mutable workspace. Where a shared filesystem is involved, the common answers are a single writer, or per-agent sandboxes with a merge step.

The claim this paper makes is modest and, as far as the pass established, not widely implemented: **the phase split.** Applying optimistic concurrency to the expensive phase (read and draft) and pessimistic locking to the cheap one (apply), within a single unit of work, is what makes the expensive part parallel without making the mutation unsafe. It is a small idea; its value is entirely in which phase gets which discipline.

### 6.4 What this paper does not claim

It does not claim the four premises in §3 are novel; each is folk wisdom somewhere. It claims that they are *load-bearing*, and offers dated, first-party incidents in which violating each one cost weeks to months on a system that was otherwise well-documented, well-tested, and actively maintained.

It does not claim the design corpus of §2 is the right structure for design documentation in general. It claims, with a measurement, that documentation invariants and liveness are orthogonal, and that a system carrying only the first will believe itself healthy while dead.

## References