Deterministic, Reproducible, and Wrong: How Our Own Scorer Manufactured an Agent-Memory Finding, and the Guard That Would Have Caught It

Swapnanil Saha
Independent
swapnanilsaha26@gmail.com

August 2026

Abstract

In the agent-memory literature we surveyed, provenance is primarily used to make agents trust retrieved context *less*: poisoning defenses, injection gating, watermarking, and transfer integrity all point in that direction. This paper asks the inverse question. When a harness delivers a legitimate, correct note into an agent's context, and the note asserts something the agent can neither corroborate from its workspace nor arrive at through its default behavior, does attaching a corroboration affordance, a way for the agent to check the claim for itself, change what the agent does where authoritative wording does not?

We report a pilot that does not answer that question, and the reason transfers further than the answer would have. Our first pass produced a clean headline: delivery of the note was confirmed from two separate sides, and the agent still committed the exact mistake the note existed to prevent. That headline was an artifact of our own scorer, a deterministic program with no language model anywhere in it. Its mistake signature matched shell text anywhere in a command, so it fired on a read, `cat deploy.sh`. The transcript instead shows the agent declining to run the script and citing the delivered note as its reason. Re-scoring every preserved session in the reported grid at zero additional model cost cleared every flagged leg.

The pilot could not answer its question, but it exposed an evaluation failure whose diagnosis, correction, and guard we consider the primary contribution of this work. What survives is smaller and, we think, more useful: an outcome-level negative result for the provenance-only note variant; one fully documented specimen of a delivered note entering an agent's stated reasoning, with the counterfactual left open; a shell matcher that fires only at execution position; and a machine-readable contradiction guard between what the agent did and what the workspace shows, which fails a record automatically when they disagree. The wrong number was never published. Catching it was ordinary quality control before submission; reporting it is not. We report a direction, a negative result, and a list of the things our own harness got wrong.

1 Introduction

Memory fails in three places and gets measured as one. Retrieval misses, so the note is absent or wrong. The note sits unread in the store (the database of saved notes) because asking for it was the model's choice and the model did not make it. Or the note is delivered, sitting in the agent's context in plain text, and the agent does the other thing anyway. We call a note *delivered* when the harness, the code that runs the agent and controls what its context contains, has placed it into the agent's context before the session's first user turn, and the word carries that sense throughout this paper.

The first failure is a retrieval problem and it is well studied. The second is an adoption problem, and our prior work reports its starkest single-session instance: an agent with connected memory tools, explicit guidance, and a store pre-seeded with directly relevant facts made zero memory calls in 114 turns [1]. One response, and the one this paper studies, is to remove the choice: deliver the memory from the harness, at the moment the situation cues it, whether the agent asked or not.

This paper set out to study what happens after that fix works, and ended up studying whether we could tell.

1.1 What we expected to report

A synthetic repository contains `deploy.sh`. It appears to work: it prints progress, exits zero, and writes a state file. It does not work. A reconciler silently reverts it, and the correct action is to append to a queue file. Nothing in the repository says so. The fact enters once, in the first session's prompt, from the user. Between sessions we plant a note carrying that fact verbatim in the store. The second session is a fresh process with no conversation carry-over, and two separate gates confirm delivery before anything is scored. The session-start hook counter in the daemon, the background process that holds the note store, incremented. And the planted note's content appears verbatim in the agent's own transcript.

Our scorer said the agent then ran `./deploy.sh` as its second tool action, with no self-correction and zero memory tool calls. We wrote that up as **delivered but not heeded**: what we then took to be the failure mode that survives every fix the field currently knows how to apply.

1.2 What actually happened

The mistake signature was `BashAction(r"(\./)?deploy\.sh\b")`, a bare regex over the command string that never checked whether the command was executed. The agent's three tool actions in that leg were all read-only. The signature matched inside `cat deploy.sh`.

The scorer had already recorded the contradiction and had no way to notice it. Its checks come in two kinds. An action-derived check reads what the agent did, from the ordered commands in the transcript. A state-derived check reads what the workspace shows, from the bytes on disk.

The same result record carried `deploy_state_untouched = PASS`, a state-derived check: the state file's hash equaled the baseline taken at the start of the leg. Running the script writes that file, so an unchanged hash means the script did not run to completion. Beside it sat two signals saying the opposite: `mistake_committed = true`, and `no_direct_deploy_script = FAIL` reporting "2 matching command(s)".

Those last two signals are not independent of each other. `no_direct_deploy_script` is a `CommandRan` check over the same defective regex family, so it is action-derived, and it agreed with the mistake flag automatically because both read the same pattern. The record therefore held two independent observations, not three. Its real shape is **one state-derived check contradicting two agreeing action-derived signals**, and nothing in the pipeline compared across that boundary.

The agent's closing message settles it (quoted with an internal punctuation elision, and with the model's own emphasis preserved):

> "I did **not** run `./deploy.sh staging` [...] per what I know from this workspace, that path appears to succeed locally but gets silently reverted by the reconciler on its next pass, leaving no trace."

Delivered and heeded, and stated as such in the agent's own words.

A previous revision of this paper got the shape of the record backwards. It described the record as two state-derived checks contradicting one action-derived check, and it reported `no_direct_deploy_script` as PASS with zero matching commands. PASS with zero matching commands is the value that key takes in the *rescored* record. The original reports FAIL with two. That description is retracted.

We fixed the signature so that it matches only at execution position, added a machine-readable contradiction guard that fails loudly when an action-derived verdict and a state-derived verdict disagree, and built a re-score path that replays preserved transcripts at zero model cost. All five flagged legs in the reported grid flip to clean. That includes the shared first session, whose mistake was supposed to be the event the repetition metric counts from. **In the reported grid, no agent in any arm ever executed the script.**

1.3 The claim, and what this paper is

The paper's sole research claim is C4 (the label is this project's own claim numbering):

> **C4.** Legitimate agent memory may be under-trusted. Attaching a corroboration affordance to a note, meaning an origin event, a date, a one-command verify hint, and a content-hashed anchor to a real file, is a candidate intervention. Its effect should be testable independently of the note's wording, its authority, and the channel that delivered it.

The last sentence of C4 is a design requirement, not an achieved property: this pilot did not meet it, for the channel because the arms were not matched (Section 3.3) and for the claim class because the uncorroborable cell could not carry the affordance at all (Section 2).

Everything else here is apparatus or defect record. The only contribution claims are the three stated in Section 4.4. The longitudinal harness, the split of facts by origin, the mistake-repetition metric, the delivery arms, and the validity gates exist because C4 could not be asked without them.

This pilot does not answer C4. It is a pilot in the literal sense: it was run to find out whether the instrument works, and the answer was mostly no, in specific and fixable ways that we enumerate. We publish it because the failure is better evidenced than the finding it replaced.

2 The inverse provenance problem

A note is **legitimate** if it is true of the workspace and was authored by the user or by a prior session under the user's direction. A note is **corroborable** if some artifact reachable from the agent's working directory would confirm it: a file, a command output, a log line, anything the agent can look at instead of taking the note's word. That property is the one the whole paper turns on. The third term is needed only twice: a note is **prior-contradicting** when the model's own default behavior, absent the note, runs opposite to what the note prescribes. "Prior" here always means the model's default, never a piece of prior work.

The hard cell of that three-way split is legitimate, prior-contradicting, and *un*corroborable. The agent is asked to abandon a strong general belief about how software projects work on the strength of a sentence in its context and nothing else. Recent work finds agents over-trust persistent memory badly enough to be a security problem [6]. Attacker payloads planted in auto-loaded instruction files persist across sessions and drive unauthorized behavior, at compliance rates reaching 100%. That work recommends explicitly that memory not be treated as uniformly trusted. On that reading,

an agent that declines to act on an uncorroborable claim someone has inserted into its context is behaving correctly.

We think both findings are real and are about different content. The payloads in [6] are *instruction-shaped*: they tell the agent to do something, typically something it is already willing to do. Ours are *claim-shaped*: they assert a fact that requires the agent to stop doing something it believes is right. Agents may over-comply with injected instructions and under-comply with injected corrections. If that holds, trust in ambient context is not one quantity, and modeling it as one will mislead. Reconciling that asymmetry empirically is, we think, more valuable than either finding alone, and it is why the C4 intervention is not simply "make memory more persuasive", which would be a security regression. The one observation in this pilot bearing on the asymmetry points the other way. The Section 6.1 specimen is an agent complying with an uncorroborable correction that the harness put into its context. That single observation is too weak to refute the hypothesis, but it does not support it.

The point is not to make the agent trust the note more. It is to give it a way to stop having to. A corroboration affordance is an invitation to check: the note says where the fact came from, when it was recorded, which file it is anchored to, and one command that settles it. If the agent takes the invitation, the note stops being a claim and becomes an observation.

This has a corollary that bounds the entire result: **the content-corroborating form of the affordance can only be built where the workspace records the fact.** On an uncorroborable scenario there is no file to anchor to and no command to suggest, and fabricating one would destroy the case we are trying to study, because the no-memory control could then discover the fact too. So the hardest version of the problem is the one place where the strongest version of the fix cannot be tested at all. More sessions will not remove that. It is a property of the design, not of the sample. Whether an affordance that corroborates the note's own origin chain instead of its content escapes the bound is open, and Section 9 records it as open.

3 Apparatus

This section describes instrumentation. Section 4 describes the parts of it that were wrong.

3.1 Sessions and trajectories

Here is the vocabulary, in one place. A **leg** is one non-interactive agent invocation, meaning it runs start to finish with no human available to answer a question; where the context is chronological we call the same thing a session. An **arm** is one experimental condition, defined by how the note reaches the agent, or by its absence. This paper has three of them, and Section 3.3 lists them. A **trajectory** is a scenario, an arm, a note variant, and a seed, run over legs k = 1, 2, 3, where k is simply the session number within that trajectory. The **grid** is the set of trajectories whose numbers Section 5 reports, and a **cell** is one arm's entry in it. The **scorer** is the program that reads a finished leg and computes its numbers, after the agent has stopped, with no model involved at any point.

Nothing carries the conversation from one leg to the next: each leg is a fresh process with no conversation resumption. Exactly two things cross a session boundary, the workspace and the memory store. Leg 1 is run once per scenario under control conditions (the store is empty in every arm, so there is nothing to inject) and reused as the starting state for every arm's leg 2. This is the largest cost saving in the design, and it also removes one source of variation between arms: every arm starts leg 2 from a byte-identical workspace. In doing so it removes statistical independence between the arms, which Section 8 records as a limitation.

Section 8 also explains why the carry-forward workspace turned out to be a serious confound and not a convenience.

3.2 Two origin classes

The two classes are capitalized throughout. They are class labels, not emphasis.

DISCOVERED facts are corroborable: the workspace records them at real cost, so a later session can re-derive them and the loss from forgetting is measurable. Primary metric: re-discovery cost. The scenario is `release_via_ci`, a package repository whose releases are cut by pushing an annotated tag. A workflow then publishes the package through a trusted-publishing flow that uses no API token. The agent's usual approach, building locally and uploading, has no credential to authenticate with and cannot work. "Usual" describes the draw toward the direct route, not a measured error rate: the control arm's measured mistake rate in the Section 5 grid is 0.0 at both legs in both scenarios. The fact is recoverable by reading the workflow file.

TOLD facts are uncorroborable: nothing in the workspace records them, the fact enters once from the user, and a later session either carried it or errs. Primary metric: mistake-repetition rate. Re-discovery cost is censored here by construction: no value is recorded because the event that would produce one cannot occur. There is nothing recorded to re-discover, so the leg carries a blank, never a zero. The scenario `deploy_reverted_by_reconciler` is the deploy script of Section 1.1.

That definition of TOLD is stronger than the scenario we actually built. The gap is in the fixture (the files we authored as the leg-1 starting workspace) rather than in anything a leg left behind. The TOLD fact has two halves: a causal half, that the reconciler silently reverts a direct deploy, and a prescriptive half, that the queue is the sanctioned channel. The workspace records nothing of the first. It does record the second, at leg-1 start, in two files we authored: `deploy/README.md` reads "queue.yaml is consumed by the release bot.", and `queue.yaml` opens with "# Consumed by the release bot. Entries are appended, never edited in place." An agent that reads those and declines to bypass the documented process reaches the correct action without ever knowing why the wrong one fails. The control's own stated reason was that running the script manually "would bypass the queue-based process this repo documents".

This route is independent of the two defects Section 8 records, the leg-1 residue carried forward (DEFECT 10) and the scenario name leaking through the working-directory path (DEFECT 11). The defects carry numbers from this project's own running record, which is why the two named here are 10 and 11. The route is present at leg 1, before any residue exists, and it does not pass through the directory name. It is a defect in the scenario fixture, and the accurate statement of the class is that the fact itself is not recorded while the behavior it prescribes is partially documented.

3.3 Arms

- **none**: no memory. Store created and asserted empty at every leg.
- **proxy**: the agent talks to the model over HTTP, and a local proxy adds the note to every outbound request as it passes.
- **hook-sessionstart**: the note is delivered once, as extra text prepended to the agent's context at session start, by a harness hook.

In tables and prose these appear as `none`, `proxy`, and `hook`, with the note variant written after a slash where it matters (`proxy` / `plain`).

Every arm at legs k >= 2 runs the agent through the same proxy process, with note insertion switched off everywhere except in the proxy arm. Insertion here is our own delivery of a legitimate note, and never an attack; where our code and counters say "injection", this is what they mean.

Routing every arm the same way keeps the network path identical between them, so the effect of inserting the note is not confounded with the effect of being proxied. No arm enforces anything: the note is advisory in every arm, and nothing in the harness makes the agent follow it.

We originally described these arms as holding content byte-identical and varying only transport. That was wrong. The two arms differ in three ways we did not intend. The hook arm plants the note with `kind="directive"`, where `kind` is the memory tool's own category for a note and governs how the note is surfaced; that choice was made to fit the session-start channel's limit on how much text it may add. The proxy arm plants the scenario default, `kind="gotcha"`, a different category in the same taxonomy. And the hook renderer wraps the note in a provenance envelope of its own, a prefix the channel adds around the note text (`[HIGH] [DIRECTIVE] ... Recorded 2026-07-30 ...`). The arms therefore differ in note kind and in delivered wrapper text, not only in transport. Any claim that the channels were matched is unavailable from this data and is not made.

The hook arm is additionally contaminated: `notes_in_store_at_start` is 2 at k=2 (with `notes_in_store_pre_plant` = 1), and ranges from 2 to 3 across its k=3 legs, against exactly 1 for every proxy leg and 0 for every control leg. The hook payload in the specimen leg reports "2 fired", and the two rendered notes are byte-identical copies of the planted note: the fresh plant plus a stale copy surviving from the prior leg's plant (Section 6.1). Hook-arm cost figures are not comparable to proxy-arm cost figures and we do not compare them.

3.4 Note variants

All variants contain the scenario's fact sentence byte-identically, asserted by test, and all name the anchor path in the body. Only the provenance trail varies.

variant	body	available on
`plain`	the fact sentence	all scenarios
`provenance`	fact, plus origin event, date, and workspace attribution	all scenarios
`verifiable`	fact, plus origin and date, plus a one-command verify hint, plus a content-hashed file anchor	corroborable scenarios only

On `release_via_ci` the provenance trail adds a sentence of the form "Established 2026-07-12 in session 1 of this workspace, after a local upload was rejected (no API token exists for this project)", and `verifiable` adds `Verify: grep -n "id-token" .github/workflows/release.yml` plus the anchor. The trail is 186 characters, written into the note body, so it is delivered by construction on any channel that delivers the body.

3.5 Metrics, and the no-judge rule

A judge, in this literature, is a language model asked to grade a run. There is none here. Every number comes from one of four sources: final workspace bytes, the ordered `tool_use` blocks of the transcript (the saved record of everything the session sent and received), the exit code of a scenario-owned verify script, or the transcript's own result and usage fields. No language model judges a run, and no number in the Section 5 grids is derived from the agent's prose. The specimen analysis of Section 6.1 is deliberately prose-derived. It reports one leg rather than a rate, and Section 4.3 explains why we no longer treat prose-blindness as a virtue in itself.

Censoring, not imputation. Censoring is meant in the statistical sense: a value is missing because the event that would have produced it did not happen, and we do not substitute one. If no acquiring action occurs, all cost components are recorded as null, the leg is marked censored, it is excluded from every cost aggregate, and no value is filled in for it. Censored legs still count in full toward mistake metrics, and the censored count is printed adjacent to every aggregate. In the tables below, "censored", "n/a", and a blank cell are the same statement. "Undefined" is a different one: it marks a rate whose conditioning event never occurred, so the rate has no denominator to be computed over.

3.6 Validity gates

A leg counts only if the arm's premise is confirmed from two sides. The proxy arm requires three things: the store is non-empty, the planted note's anchor path appears in an injection audit event logged after the note was planted, and the proxy's own injection counter is positive. The hook arm requires the store to be non-empty and the daemon's hook-injection counter to have incremented, and additionally requires the planted note's content to be found in the transcript (valid because the session's saved record includes the extra text a hook adds at session start). The control arm requires an empty store, zero injection events, zero proxy injections, and no memory tools connected. Any failure marks the leg invalid with a reason.

Two choices carry most of the weight. **Delivery evidence is required from both the daemon side and the transcript side** wherever the delivery channel allows both, and that pairing is what keeps the Section 6 specimen from being an anecdote. The two sides sit on one delivery path and are not independent draws, but they are not redundant either: the daemon-side counter counts every hit on the session-start endpoint, including the harness's own preflight probes, so on its own it cannot prove agent-visible delivery, and the transcript-side containment check is what does (Section 6.1 enumerates the three hits behind the specimen's counter value of 3). And **whether the agent uses the note is the measured outcome and is never a gate**: a design that gates on behavior cannot observe the failure this paper is about.

Before any paid session, a probe fetches the planted note through the arm's own channel and aborts the leg if the note does not come back. Four would-be-invalid legs were caught this way at $0.

4 Instrument validity: how a judge-free scorer manufactured a headline

This section is here because our scorer was wrong, and the shape of the error is reusable.

4.1 The defect

Shell-action signatures in the scorer were bare regexes over the command string. `BashAction( r"(\./)?deploy\.sh\b")` matches `./deploy.sh staging`, and it also matches `cat deploy.sh`, `stat deploy.sh`, and `ls -la deploy.sh`.

The counts, with denominators, because a previous revision of this paper got them wrong. Across all thirty legs preserved at the time of the audit, **seven** carry `mistake_committed = true`. **Five** of those seven are in the reported grid: the shared first session, control at k=2 and k=3, hook-sessionstart at k=2, and proxy-provenance at k=3. Every one of the five is a read, in every arm including both control legs. The remaining **two** sit in directories the paper does not report, one labeled poisoned and one labeled superseded, our own names for a run whose setup was spoiled and a run

replaced by a later configuration. Run directories carry the seed as `s0` and the disposal label after a dot; the full scheme is under Artifact availability. The poisoned leg is another read. **The superseded one is not.** In `deploy_reverted_by_reconciler-proxy-plain-s0.superseded-legspace-*` at k=2, the agent's action at index 4, the fifth tool call, is `echo "--- running deploy ---"` followed by `./deploy.sh staging`, and that leg's `deploy_state_untouched` is FAIL with a hash mismatch. The script really ran.

We keep that leg in the record because it is the only evidence that the corrected signature still fires on anything, and a matcher that never fires proves nothing. This one fires on the genuine execution at action 4 and rejects both false matches in the same leg, a quoted `echo` label and the `cat deploy.sh` read, both inside the compound command at action 1. Discriminating inside a single transcript is the evidence we have that the corrected signature still catches what it was built to catch.

The regex was too loose. The more serious failure is the second one: **the scorer's own record contained the refutation and the scorer could not see it.** In every flagged original, one state-derived check, `deploy_state_untouched = PASS` from a `FileUnchanged` comparison against the leg-start baseline, sat beside two action-derived signals that agreed with each other and disagreed with it: `mistake_committed = true` and `no_direct_deploy_script = FAIL`, the latter reporting "2 matching command(s)" from the same defective pattern. The contradiction was machine-visible, in one JSON object, and nothing crossed the action/state boundary to look at it.

4.2 The fix, and the re-score

Three changes, all in the harness:

1. **Execution-position anchoring.** Mistake signatures, and the acquisition signatures that decide whether the agent got the fact, now match a shell pattern only when the script is the thing being run rather than merely named. The matcher handles `./`, `bash x`, `cd a && x`, `;` separators, pipes, and `env`/`timeout` wrappers, and it rejects `cat`/`stat`/`grep`/quoted-echo mentions. It is a heuristic over an enumerated set of wrappers, not a shell parse. The one real execution we have, from Section 4.1, played no part in building it, and it still catches that one. It also passes a 23-case adversarial probe we wrote ourselves (six reject and eleven accept cases for the deploy pattern, three and three for the git-tag pattern, which is the whole probe). The probe cases were written together with the fix, after the defect was diagnosed, in the same commit; they are a regression suite, not independent validation, which is why the held-out execution above carries the evidential weight.
2. **A machine-readable contradiction guard** between action-derived and state-derived checks, so that a future disagreement of this shape surfaces in the result record instead of being silently resolved in favor of the regex. The guard is one-directional by declared scope: it fires when an action-derived mistake meets a clean state check. An execution the matcher misses produces no mistake flag and therefore no contradiction; the state-side check fails on its own, but nothing links the two in that direction, so a false negative of the matcher is invisible to the guard.
3. **A zero-cost re-score entrypoint.** Because every transcript is preserved, the corrected scorer replays them and writes `result.rescored.json` next to the immutable original. No model spend, no re-running of a completed experiment, and the original record is never overwritten.

The fix is merged to the harness's main branch. The re-score is byte-identical to the original on the DISCOVERED scenario, flips all five flagged TOLD legs in the reported grid to clean, and fires

zero contradictions. Only the twenty legs of the reported grid were re-scored; the two flagged legs in superseded and poisoned directories were not, which is why the genuine execution of Section 4.1 still stands in the record as a mistake.

4.3 What this costs the paper, and what it buys

It costs the headline and the entire TOLD half of the results.

First, a statement of scope for judge-free evaluation. "No language model in the judging path" is a real property and we still recommend it, but it is a claim about *who* judges, not about whether the judgment is correct: a deterministic scorer is exactly as wrong as its patterns. It is wrong silently and reproducibly, and its reproducibility made our wrong number look more trustworthy instead of less. Our first draft asserted, as a virtue, that no agent prose was read. The defect was found by reading agent prose, which is why Section 3.5 now scopes the rule to the reported grids instead of to the whole paper.

Second, a design rule we now follow: **where a fact is observable both from actions and from state, compute both and make disagreement an error.** The information needed to catch this was already being collected; only the comparison was missing.

4.4 What is new here, and what is not

The general finding is not ours and we want to be explicit about that, because 2026 has been a heavy year for it. A validity audit of tool-calling evaluation puts evaluator-against-human misalignment at 18.5% over 496 expert-reviewed tasks, and names substring-based matching and what it calls brittle state matching as deterministic-evaluator failure modes [18]. Automated benchmark auditing finds more than a quarter of tasks flawed across 168 benchmarks, and filtering them shifts model rankings [21]. Automated transcript analysis extends the same program with scanners for four validity-flaw classes, among them ground-truth access, the leakage class of our DEFECT 11 [41], and failed trajectories have been compiled into structured evidence for diagnosing and repairing harness flaws [42]. Log analysis has been argued to be necessary for credible agent evaluation at all [20]. On the memory side, a taxonomy of agentic-memory evaluation catalogues metric defects including lexical-overlap penalties and what it calls backbone-sensitive silent failure [24], and an independent audit of a widely used long-term-memory benchmark found a 6.4% answer-key error rate [29].

The one that documents our exact contradiction class in recorded data is an evidence-bounds audit of interactive agent benchmarks [19]. Scalar rewards, it observes, "accept failed required actions, wrong state, or inconsistent DB criteria", and its repair is to make required subchecks gating conditions for the reported score, so a run with a failed required check cannot be reported as a pass. Closest in shape to this section is a self-audit that traced an apparently robust 32-point accuracy gap to a shared `max_tokens` parameter silently truncating its own generated data [17]. Their number had replicated at N=50 and tightened at N=500 without changing direction, and they write that they treated that stability as evidence against measurement noise, correctly "in the narrow sense that the underlying number was indeed stable, though a different, undetected problem remained". We take that as the general lesson stated in its strongest form, and we do not restate it as ours. Their fault sits upstream in a generation pipeline; ours was in the scorer. Their corpora are synthetic LLM-judge corpora, not agentic action traces. And they caught it by reading, which is also how we caught ours.

Three specifics appear to be unclaimed.

First-person self-refutation. Every instance above is an external audit of someone else's benchmark, or a methods position paper. The one self-disclosure we located is a changelog entry

in an unrelated education paper reporting a multi-turn scoring misalignment [23], not a paper organized around its own refutation. Adjacent genres exist and deserve naming rather than an unqualified absence claim. Biomedical publishing operates retract-and-replace as standing policy for pervasive honest error whose correction changes the results while the science remains reliable [47]. Psychology's Loss-of-Confidence Project collected researchers' own accounts of losing confidence in their published findings and reports the sentiment is common but rarely public [46]. And a 2026 introspection paper states in its Section 3.4 that the magnitudes it previously reported "are all artifacts of the split and ordering defects documented here" and prints the corrected values in the same document [48]. The first replaces the document, the second surveys private doubt, and the third is a correction section inside a paper organized around a different result. What we are not aware of remains narrower: a paper in agent or language-model evaluation whose organizing result is the reversal of its own prior headline with the corrected numbers in the same document. The refuted headline was never published. It existed in an internal draft, and catching it is ordinary quality control before submission. What we think is unusual is not the catching but the reporting, since the normal outcome of that quality control is that the reader never learns the wrong number existed, and here the wrong number and how it was produced are the paper.

A contradiction guard that needs no human. This claim has to be narrowed against the nearest prior work, and our earlier phrasing did not narrow it enough. Reference [19] already flags disagreement automatically. It surfaces a record whenever the benchmark's own label disagrees with an evidence-derived label, with no human choosing which records to surface. Only the *adjudication*, the step where a flagged record is judged, is human, and the paper is explicit that "reviewers inspect records flagged by native/evidence disagreement" before aggregation, with subcheck gating recommended as the repair. Automatic detection of disagreement is therefore not ours. What is left is narrower. The disagreement we detect is between action-derived and state-derived checks inside a single deterministic scorer, rather than between a benchmark-native label (the benchmark's own recorded answer) and an evidence label derived with a model's help. There is no model anywhere in our path, at detection or at adjudication. And the consequence is automatic failure of the result record instead of routing to a reviewer.

The technique itself is older than any of this and we cite its lineage rather than imply it is new. Computing the same property along independent routes and treating disagreement as a defect signal is differential testing [37]; the variant in which the computations are linked by a known relation is metamorphic testing [38]; building the redundancy in at design time is N-version programming [39]. All three exist because trustworthy oracles are scarce, which is the same reason our scorer had no one to check it. The guard is an application of a differential oracle to a scorer's own outputs, and what we claim is the pairing: the action-derived and state-derived views of one agent trace are cheap to compute together, and their disagreement is machine-decidable with automatic consequence and no model.

Execution-position anchoring. Matching a forbidden command at execution position, instead of anywhere in the command string, is a small and specific fix that we did not find named in the evaluation literature. The nearest prior work is the "substring-based" failure category of [18], and on inspection of that paper's own worked case the category covers substring assertions over an agent's natural-language final response, not command-line action matching. The two are different failures of the same shape, and we claim only the second. Outside evaluation the problem is old and the security literature is further along: shell allowlisting and sandbox policy engines have long distinguished the program being run from a string that names it, and a 2026 security analysis of an agent framework demonstrates that lexical matching over command text cannot enforce a semantically defined boundary, with bypasses from line continuation, multiplexer dispatch, and option abbreviation [40]. Our matcher is a regex anchor by comparison, and the claim is only the

transfer of execution-position discipline into deterministic scorer signatures, where the failure mode is not an attacker but the scorer's own author.

We also note, as a protocol rather than a result, that re-scoring preserved transcripts costs no model spend. Prior work argues for log analysis [20] and for treating scores as perishable claims requiring prospective controls [22], but neither operationalizes cheap retrospective correction of numbers already reported. Preserving transcripts is what made re-scoring every run in the reported grid a $0 operation here, and we recommend it on that basis alone.

Finally, a caution that applies to this paper's own framing: judge-free state-based scoring is not our idea and we do not present it as a virtue we introduced. VehicleMemBench [4] already claims objective and reproducible evaluation without model-based or human scoring by comparing post-action environment state to a target state. Section 4 is precisely an argument that this property is worth less than it sounds.

5 Results

Every cell is N=1: one observation, no repeat. Two scenarios, one seed, one agent model, legs k = 2, 3 per trajectory; a second seed, run later on the repaired harness, is reported separately in Section 5.4. Numbers below come from `result.rescored.json` where it exists and `result.json` otherwise. Total spend is $17.54 across all 58 preserved legs, of which $6.34 is the reported grid (the 18 scored legs plus the two shared first sessions); the remainder is the second seed, harness-fix verification legs, and superseded and invalidated runs, retained. The grid figure reconstructs from the tables: the session-cost columns of Sections 5.2 and 5.3 sum to $3.778 and $2.067 (unrounded; the printed cells give $2.066), and the two shared first sessions cost $0.300 (DISCOVERED) and $0.196 (TOLD), totaling $6.341.

5.1 Three broken metrics

Our `billable_tokens_to_fact` aggregate is defective and we do not cite it, because a part cannot exceed the whole it belongs to and this one does. Its running total exceeds the whole session's own token count in every uncensored leg, by up to 2.6x (340,907 against 132,567 for the hook leg at k=2). The metric is meant to count the tokens billed up to the moment the agent acquires the fact. It computes that by adding a billing-weighted usage figure for every assistant event from the start of the session up to that moment. The transcript emits several assistant events per API call (33 events across 13 calls in that same hook leg), so each call's usage record is added more than once: the same weighted sum taken over the whole session reaches 354,186 against the session's own 132,567. The excess is duplicated per-event usage records, not legitimate cache accounting. The difference the metric appeared to show between arms tracks how much of each session's context was newly cached instead of reused (k=2 cache-creation tokens: 13,479 verifiable, 20,977 control, 23,875 proxy-plain, 42,653 hook, the exact order of their `billable_tokens_to_fact`; the k=3 legs do not preserve that order, so this is a k=2 observation). That is a property of the prompt cache, not of the agent.

We therefore report `context_tokens_at_fact`, the size of the context the agent is holding at the moment it acquires the fact. That number is recorded in the same result object, it is not double-counted, and it points the other way.

A third metric, `turns_to_fact`, counts assistant events rather than conversational turns. An assistant event is one message from the model, and one conversational turn can contain several. In every leg the metric exceeds the session's own turn count (for example 33 against 20 in the verifiable leg at k=2). The design pre-registered its "turns_to_fact <= 2" expectation in conversational-turn units (pre-registered here means written down and frozen before any session ran). Our first draft

compared the two unit systems directly, which overstated the size of the miss by roughly 2 to 3x depending on the cell. The miss is real either way. Recounted from the preserved transcripts in conversational units (distinct API calls up to the acquiring action), the uncensored DISCOVERED cells read: control 13 / 11 at k2/k3, proxy-plain 11 / 13, hook 11 at k=2, verifiable 15 / 13, against the pre-registered 2 or fewer, a miss of five times or more in the metric's own intended unit. The two turn units also disagree with each other about direction: in conversational units the hook arm reached the fact faster than control at k=2 (11 against 13), the cell the assistant-event unit calls a tie. The unit is corrected here, and the metric should be renamed.

5.2 DISCOVERED: `release_via_ci`

arm / variant	turns to fact (k2/k3)	context tokens at fact (k2/k3)	release outcome (k2/k3)	session $ (k2/k3)
`none`	30 / 29	40,568 / 39,649	pass / pass	0.353 / 0.317
`proxy / plain`	27 / 29	43,901 / 41,023	pass / pass	0.402 / 0.358
`hook / plain`	30 / censored	42,417 / n/a	pass / **fail**	0.494 / 0.285
`proxy / provenance`	censored / censored	n/a / n/a	**fail / fail**	0.407 / 0.359
`proxy / verifiable`	33 / 27	45,342 / 41,551	pass / pass	0.434 / 0.369

"Release outcome" is the `annotated_tag_exists` check: whether the session actually produced a correctly annotated release tag. Mistake rate is 0.0 in every arm including control, so the error axis is uninformative on this scenario. No agent made the mistake the scenario was built to invite.

`context_tokens_at_fact` is defined in five cells, three memory arms at k=2 and two at k=3. In every one of them the memory arm reached the fact with more context than the no-memory control, and control is the cheapest cell on that measure in both legs. The provenance arm contributes no cell because it is censored on this measure: it acquired the fact by a route the acquisition signature does not match, and the transcripts show it had the fact anyway (Section 8, under Construct validity). We previously stated this as a universal over memory arms, called it "the clearest signal in the grid", and said memory "lengthened the path to the fact". All three overstate it, for two reasons.

First, the measures disagree. Five cells support a turn comparison against control, both arms uncensored, and in assistant-event units memory is faster in two of the five (proxy-plain, 27 against 30 at k=2; verifiable, 27 against 29 at k=3), tied in two (proxy-plain at k=3, 29 against 29; hook at k=2, 30 against 30), and slower in one (verifiable, 33 against 30 at k=2). Any claim that memory lengthened the path is measure-dependent and we withdraw the unqualified form.

Second, the delivered note is itself part of the context the agent is holding when it reaches the fact. The measurement therefore includes the very thing it is supposed to be evaluating, which is an arithmetic confound and favors the control automatically. On this scenario the hook delivered 1,269 characters at k=2 and 1,593 at k=3 (the payload carries the contaminating duplicate note, Section 3.3), roughly 320 and 400 tokens by a four-characters-per-token estimate, about a sixth of the hook cell's own 1,849-token gap at k=2. The proxy channel injected about 735 characters, roughly 180 tokens against gaps ranging from 1,374 to 4,774. The payload therefore accounts for a minority of every gap, and what produced the rest is not decided by this data.

What survives is narrow. On this measure, in this pilot, delivering a correct note did not pay for itself in context. Whether that is reconciliation cost, distraction, the payload's own arithmetic, or N=1 noise, this data cannot say. Separating those is the first thing more sessions would buy.

This observation is confirmatory, not new, and we cite the earlier result explicitly. A budget-constrained study of web agents reports that a token-matched vanilla baseline "matches or surpasses all three augmentation methods in aggregate success rate while often using fewer total tokens" across three WebArena domains and three models [25]. That result predates ours, is far better powered, and already establishes the general point that memory augmentation can cost more than it saves. Our contribution on this axis is nothing more than a second setting, coding agents, across session boundaries instead of within one session, and we headline no cost claim.

The provenance-only variant failed the task outright in both later sessions, and the reason is not the one the bare outcome suggests. It is the only configuration that never produced a correct annotated tag at either k, while `none`, `proxy/plain`, and `verifiable` completed six of six between them. Our first draft buried this by reporting the trajectory only as "censored". But reading the transcripts changes the interpretation substantially, and we report the texture because the bare outcome is misleading.

Both provenance legs passed every sub-check, meaning every individual check that feeds the leg's score, except `annotated_tag_exists`. Both stated the trusted-publishing fact correctly in prose. They did not fail to tag through ignorance or error. They explicitly declined to tag, on the ground that tagging is irreversible, and asked the user for permission. At k=2 the agent set out the consequence, that "tagging and pushing `v1.4.1` will trigger a **real, public PyPI release**, which isn't reversible", listed what it would need, and closed with "I'll hold off on tagging/pushing until you confirm." At k=3 it wrote that pushing a tag "is a real, effectively irreversible action once a remote exists, so I didn't create/push a tag", and then "Let me know if you want me to add a remote and push a `v1.5.0` tag." The control and verifiable arms ran `git tag -a` without asking; the provenance arm never ran anything but `git tag -l`.

This exposes a confound we did not design for: **a non-interactive harness converts permission-seeking into task failure.** The agent had nowhere to send the question. On a metric that scores final workspace state, an agent that became more cautious scores identically to one that became less capable.

The two legs are not independent observations: k=3 starts from k=2's un-tagged workspace, so what the grid shows is one failure and its dependent sequel, not two. And the shared first session of this scenario also failed `annotated_tag_exists`, under control conditions with no memory at all, which means the check is not trivially passed even in the baseline. The k >= 2 finding stands as stated, but 0 of 2, being one failure and its sequel, is less remarkable against that background.

We searched for a prior report of a legitimate provenance trail degrading task success and did not find one. The nearest results run the other way or on a different axis: source-authority framing changes action selection when propositions compete [5], and memory content is over-trusted to the point of being a security problem [6]. Neither reports honest provenance hurting. We record this as an outcome-level observation at N=1, with the permission-seeking confound attached, and explicitly not as an effect. The hook arm also failed at k=3, but the hook arm is store-contaminated (Section 3.3) and we do not stack the two.

The `verifiable` variant did nothing in particular. It completed the task in both legs, like plain and like control, and used the most context of any arm at k=2. It is not the "only" anything: `proxy/plain` is also uncensored in both legs at 27 and 29 turns. A sentence in our first draft claiming the affordance variant was the only cell to state the fact in both later legs was simply false against our own table. Dropped.

No claim about the mechanism. We can say nothing about whether the affordance worked the way we said it works. Our first draft reported `anchor_checked = true` for `verifiable` and null for `plain`, and read that as the agent taking the invitation to check. The flag is computed only for the `verifiable` variant, so "null in plain" was a measurement gate, not a behavior. Worse, the

anchor is `.github/workflows/release.yml`, which is this scenario's discovery artifact: the design's own control-arm expectation is that a no-memory agent re-reads exactly that file. Recomputing the scorer's own rule across all ten legs finds the anchor inspected in every arm including control. The pilot has no evidence that the affordance induced checking, and the metric needs an anchor separable from the discovery artifact before it can produce any.

The one uncontaminated affordance measure is negative, and we report it. Alongside `anchor_checked`, the scorer computes `verify_command_ran`, which asks whether the agent executed the note's suggested verify command. Unlike the anchor, that command has no reason to be run by an agent that is not following the note. It is `False` in both `verifiable` legs. The affordance's most literal invitation was one command, written into the note body, on the one scenario where it could actually be checked, and neither leg ran it.

That number carries two caveats. It carries the same variant-gating caveat as `anchor_checked`, since a plain note has no verify hint to run, so `verify_command_ran` cannot support a contrast between arms. And the verify command greps the anchor file, which was inspected in every arm, so an agent that had already read it had already corroborated the claim by a cheaper route. A `False` here is consistent with corroboration as much as with indifference. Testing C4 properly needs a verify command whose result is not otherwise obtainable. Within those limits this is the number in the grid that bears most directly on C4, and it does not support the intervention.

5.3 TOLD: `deploy_reverted_by_reconciler`

Every row of this table is identical, and the two paragraphs under it explain why. "Primary check" is this scenario's own outcome check, the counterpart of "release outcome" in Section 5.2: it requires the queue entry count to increase over the leg.

arm / variant	mistake rate (k>1)	repetition rate	primary check (k2/k3)	censored	session $ (k2/k3)
`none`	0.0	undefined	fail / fail	2	0.326 / 0.234
`hook` / `plain`	0.0	undefined	fail / fail	2	0.217 / 0.166
`proxy` / `plain`	0.0	undefined	fail / fail	2	0.422 / 0.203
`proxy` / `provenance`	0.0	undefined	fail / fail	2	0.327 / 0.171

Post-re-score, no agent in any arm ever executed the script, in any leg, including the shared first session. `mistake_repetition_rate` is undefined everywhere because it is conditioned on the first session committing the error and the first session did not, so the mistake axis is empty.

The primary check fails in every arm and leg for a different reason, and the reason is a design defect rather than agent behavior: leg 2 begins with leg 1's queue entry already present, the check requires the count to increase, and every agent correctly concluded that the deployment was already queued and that appending a duplicate would be wrong. Section 8 treats this as an open design question.

The empty mistake axis should not be read as evidence that agents are naturally cautious around destructive commands. Every agent in this scenario had at least three routes to the correct action that did not require the note. The fixture documents the queue as the sanctioned channel (Section 3.2). The scenario slug (the short scenario name carried in the working-directory path) names the reconciler outright (DEFECT 11, Section 8). And from k=2 the workspace carries leg 1's queue entry (DEFECT 10, Section 8). A null under those conditions is what the design should have predicted, and it is a statement about the scenario rather than about the agents.

This half of the grid says nothing about memory.

5.4 A second seed, run on the repaired harness

After the grid froze, the TOLD-scenario repairs merged: the leg-start residue reset (DEFECT 10, Section 8), opaque workspace paths with a synthetic committer identity (DEFECT 11, Section 8), and a fixture that no longer documents the prescriptive half of its own hidden fact. We then ran a second seed across both scenarios: `none` and `proxy` arms only, plain variant, the same agent model, legs k = 2, 3, $2.67 including the two shared first sessions. These eight legs are not poolable with the grid above. The grid ran before the repairs and seed 1 ran after them, so the TOLD rows in particular measure a different scenario surface. We report seed 1 as its own observation set, under every caveat of this section: each cell is one observation, and eight legs support no rate.

On the DISCOVERED scenario, seed 1 replicates seed 0's direction: all four cells pass the release outcome with `fact_used` true.

On the TOLD scenario, the repaired surface changes two things. The mistake axis is no longer empty: at k = 2 the control agent executed the deploy script directly and mutated deploy state, which is the seeded mistake the original fixture never elicited. The shared first session did not commit it, the k = 3 control leg did not repeat it, and neither proxy leg committed it. One mistake in one control leg is a direction rather than a rate, but it is the first evidence that the scenario premise can establish at all, which the screening item in Section 9 asks candidate scenarios to demonstrate before graduating to a grid. And the primary check now fails behaviorally rather than by design: after the residue reset the check asks whether the agent appended at least one queue entry within its own leg, and no agent in any of the four cells did. That includes the two proxy legs with verified delivery, one injection each with the planted note retrieved into the transcript, and `fact_used` false in both. On the one surface where the fixture no longer hands the agent the sanctioned channel, delivered memory did not become action. Whether the delivered note is why the proxy legs avoided the deploy mistake the control committed is the same counterfactual Section 6.1 could not close, at the same N.

6 What the pilot supports

6.1 The specimen: delivered, and heeded, with the counterfactual open

The most informative single leg is the TOLD scenario, hook arm, plain variant, k=2. The rescored record carries all of:

- `valid: true`, every gate passed;
- `hook_injection_counts: {SessionStart: 3}`, the daemon-side counter fired. The three hits enumerate from the audit log as the harness's plant-reachability probe, the harness's hook preflight, and the one injection in the agent's own session; the counter counts endpoint hits, the payload header's "2 fired" counts notes rendered per hit, and the next line, not this counter, is what proves the one agent-visible delivery;
- `planted_content_in_transcript: true`, the note appears verbatim in the agent's own transcript;
- `mistake_committed: false`, `first_mistake_action_index: null`, `contradictions: []`;
- `no_direct_deploy_script: PASS`, `deploy_state_untouched: PASS`;
- `vectr_tool_calls: 0`, no memory tool involved in either direction.

The leg is `censored: true`, because no acquiring action occurred, so it contributes nothing to any cost aggregate. Its primary check also failed, for the design reason common to every arm on this scenario, which makes the leg a specimen rather than a win.

We do not claim the note caused the behavior, and we no longer claim the note was the only possible source of the agent's reasoning. A previous revision said the reason the agent gave "could only have come from the delivered note". That is refuted by this paper's own DEFECT 11: the scenario slug reaches every arm through the working-directory path, including this one. The control arm, with an empty store, reached the same action, and its transcript shows how. It reasoned from the queue file and its README, and wrote that "given this environment is explicitly testing reconciler behavior, a manual out-of-band deploy is likely to get flagged as drift and reverted". The word "reconciler" appears nowhere in the workspace files. It reached the control agent through its own path, which contains the scenario slug `deploy_reverted_by_reconciler`; the transcript shows the agent picking the term up and then sweeping the workspace for it with a compound `find` including a `-name "*reconcil*"` clause, and our no-memory control was told the answer by the directory it was standing in.

The specimen's value is not that the note was the only route to the conclusion. It is that the agent's stated reason tracks the note's wording rather than the slug's. The planted note reads that running the script directly "appears to succeed but the reconciler reverts it on its next pass, and the reverted deploy leaves no trace in this repository". The agent wrote that the path "appears to succeed locally but gets silently reverted by the reconciler on its next pass, leaving no trace". Three phrases track the note: "appears to succeed", "on its next pass", and the no-trace clause. The delivered payload contained nothing else to echo: its "2 fired" header covers two byte-identical copies of the planted note (the second is the stale prior-leg plant of Section 3.3), so the phrase check below covers every sentence the channel delivered. Each of the three phrases occurs zero times in the workspace files and zero times in either control transcript. The slug carries one word, `reconciler`. The slug-only control produced only "flagged as drift and reverted", a plausible general inference that shares none of the three phrases. Two control transcripts are a thin base rate, so we report the overlap as an observation. It is not proof of what caused the wording: the slug does not account for it, and whether the note's wording or the situation produced it is not separable at this N.

The specimen supports one claim. A harness-delivered note can enter an agent's explicit reasoning at the level of its own wording, and be cited as the basis for declining a default action, observable from the transcript alone. This is not an arm contrast. The control restrained itself too, so restraint is not attributable to the note, and the specimen survives only as an instance of an agent explicitly citing delivered memory. It does not support the claim that the note produced the behavior. That would need a control without independent routes to the same conclusion, and Sections 3.2 and 8 show ours had three.

Nor is the specimen novel as a claim about hooks. That harness-injected context at high recency raises compliance is practitioner-established: a terminal coding-agent system report describes delivering reminders as user-role messages at maximum recency and observing "noticeably higher compliance rates" [26], TriggerBench [2] motivates the same move, and TRACE [3] beats advisory delivery outright with enforcement. We report the leg because it is the corrected reading that replaces the refuted one, produced by the fixed instrument from the same preserved bytes, and because the agent's own stated reason is the corroboration that this paper argues delivered memory ought to carry.

6.2 A prior pre-registered experiment, and voluntary-access failure

In July 2026 we ran a two-arm A/B on a single seeded task in a large open-source Java codebase, manipulating exactly one variable: whether one operational note was present in the store. Its arms, its metric, and the rule for reading the result were written down and frozen before any session ran. That rule returned **not supported**. Both arms scored the maximum on the primary metric (honest

verification 3 of 3 in each), so the axis the experiment was designed to test had no room left to move in.

Three sessions ran per arm. The triples below are memory arm first, control second, and all of them come from that experiment's own deterministic transcript parser, whose definitions are frozen with the pre-registration. As exploratory secondary counts, with no direction claimed: false-pass verification events occurred only in the control arm (0, 0, 0 against 0, 7, 3), build-tool invocations were 1, 3, 4 against 2, 9, 4, and one control session thrashed.

Most relevant here: one memory-arm session never accessed its available note and stayed clean anyway. That is a live specimen of voluntary-access failure, where the note is available and the agent never asks for it, invisible to the outcome metric, and it is the direct ancestor of this pilot's instrumentation choices. That experiment is complete and will not be re-run. Re-running it after seeing its result would break the frozen rule, which is the classic optional-stopping problem: a result read after an unplanned extension is no longer the result the rule was written to produce.

Voluntary-access failure itself is not offered as a contribution. Our own prior work published its starkest form [1], VehicleMemBench concludes that autonomous construction and retrieval rather than tool execution is the primary bottleneck [4], and the phenomenon is common knowledge among memory vendors. The reason it appears here is the next subsection's point: it is invisible to the metric the field would naturally use.

6.3 Why scoring against an oracle condition would not have helped

The natural instrument for "does memory help" is a ratio between two scores. One is the agent's score when the information is supplied to it outright, an oracle condition. The other is its score when the agent is itself responsible for retrieving that information. VehicleMemBench [4] formalizes this as `MemoryScore` and uses it to argue that autonomous construction and retrieval, rather than tool execution, is the primary bottleneck. That paper also independently argues for state-based, judge-free scoring. We cite it as agreeing with us on that point, since it is no part of our case against the ratio.

Such a ratio cannot tell three failures apart. The note was never asked for. Or it was asked for and the wrong thing came back. Or the right note came back and the agent ignored it anyway. All three move the ratio the same way by the same amount. And when both scores reach the maximum the task allows, the ratio of two equal numbers is 1.0 and reports no problem, which is exactly the case we hit in Section 6.2. What makes the Section 6.1 leg legible at all is observing availability and behavior on separate channels: availability from delivery counters and transcript containment, behavior from the ordered action stream and final bytes. That separation is apparatus, and Section 4 shows it is necessary but nowhere near sufficient.

7 Related work

Longitudinal and multi-session memory evaluation. We claim no first. Two works sit close enough to priority to be worth spelling out. MemoryArena [7] predates this work and owns multi-session agentic tasks where memory must guide later action, with agents distilling their own experience into memory across interdependent subtasks. Its metrics are success oriented, and it carries no cost or repetition metric and no split by fact origin. A longitudinal evaluation instrument published days before this paper [9] shares the vocabulary of our evaluation's name and is a different kind of thing: roughly 380 questions with gold answers, a language-model judge, and five memory architectures over three-week and nine-week horizons.

The rest of the neighborhood is quicker to state. A public industry benchmark [8] measured controlled cross-session memory cost for coding agents in March 2026 across three arms including a pre-written context file, reporting 15 to 28% cost reduction and 28 to 40% fewer turns on complex tasks with quality unchanged; it is vendor-authored at N=9 and it precedes us, so any "first controlled measurement of memory's cost saving for coding agents" claim is unavailable and we do not make it. The cost of failed runs has been quantified from the METR public run dataset, 24,008 runs across 21 frontier models on RE-Bench: failed runs account for 90.2% of dollars and 59.2% of tokens, with a median failed-to-success token ratio of 113x (Section 1 of [10]; its Appendix E.3 holds the per-task breakdown), and the narrative-side loss is named the Storytelling Tax. That is the nearest ancestor of our cost component, measured across independent runs rather than sessions of one agent. PROJECTMEM [11] ships a coding-agent memory layer with a pre-action repeat-failure gate, and says plainly that its numbers are usage estimates rather than a controlled benchmark, which is more candor than this niche usually manages.

Evaluation validity for agents. The category is established and crowded [19, 20, 21, 22, 23, 29], and Section 4.4 states our position in it in detail. Two entries there are worth naming here: audits of the audits themselves [28], and the position that evaluation scores are perishable knowledge claims requiring validity windows [22]. Memory-specific instances include a taxonomy of agentic-memory evaluation limitations [24] and an independent audit that found a 6.4% answer-key error rate in a widely used long-term-memory benchmark [29]. The closest work in shape is a self-audit that traced its own replicated, apparently robust effect to a truncation artifact in its generation pipeline, after scale-up had tightened the number without changing it [17]. Separately, harness design has been argued to be an experimental variable and not an implementation detail, in the sense that the harness changes agent belief [27]. Our DEFECT 11 is a concrete and unflattering instance: our harness leaked the answer to the control arm through a directory name.

Corrections compiled into enforcement. TRACE [3] is the most closely related work and its abstract opens on our motivating problem: a correction remembered in one session may still be violated in the next. It compares user preferences in two representations, natural-language rules in context against compiled runtime-enforceable checks with verifiers: an advisory memory baseline leaves 57.5% of applicable checks unmet, while compilation with enforcement takes violations from 100% to 37.6% on in-distribution coding tasks, 2.0% out of distribution, and 60.5% on memory-intensive tasks. It is better-resourced than this work and lands six weeks earlier. TRACE varies representation and enforceability, with both arms in context; we vary transport and initiative with no enforcement anywhere. Enforcement is a complete answer wherever a claim can be compiled into a check, and our TOLD class is precisely the set of claims that cannot be, because there is nothing to check against. We note that our first draft attributed to TRACE a statement that trust is irrelevant in its setting; we could not locate that statement in the paper and it is withdrawn.

Knowledge conflict. The behavioral core of C4's premise, a model holding a prior that contradicts what its context asserts, is the knowledge-conflict literature's context-memory conflict, surveyed under a three-way taxonomy of context-memory, inter-context, and intra-memory conflict [30] and benchmarked at scale [31]. Context-aware decoding shows the resolution can be steered toward the context side by a decoding-time contrast alone [32]. Cross-source conflict between textual and knowledge-graph evidence has its own benchmark [33], and context compliance under conflict has been made inspectable at inference time by eliciting the contextual and prior answers separately and recording the resolution [34]. That last diagnostic is the nearest read-time machinery we located anywhere, and the difference from C4 is the object and the actor: it diagnoses retrieved context for an observer, where C4 proposes an affordance the agent itself acts on, attached to a persistent memory whose claim is about the workspace and is checkable there. What the agent-memory

setting adds to the QA setting is persistence across sessions, an action rather than an answer as the outcome, harness-side delivery, and a legitimate rather than adversarial claim.

Provenance, and its uniformly downward direction. Concurrent work audits provenance sensitivity in agent action selection [5], holding task, proposition, position, and policy fixed and varying only source authority. Its scale is summed target-token log-odds, a within-design contrast rather than a probability, and the direction is that trusted competing propositions depress support for the correct action more than untrusted ones: the trusted-minus-untrusted removal effect in its source-only control is +1.150 log-odds, and switching a competing claim's label from trusted to untrusted flips 4.0% of targets from wrong to correct against 1.2% the other way. It has no no-provenance baseline, so it cannot say whether any label raises or lowers adherence against unattributed content. It states explicitly that verification hints, cryptographic anchors, date stamps, and command-execution corroboration are not tested, which makes it close to an ideal precursor: it de-risks the premise without claiming the intervention.

The rest of the agent-memory provenance literature is defensive: content-addressed identifiers and signatures for transfer integrity in portable agent memory [12], reliability-conditional trust caps by source [13], attribution watermarking [14], and influence-provenance graphs that gate execution on justification by trustworthy evidence [15]. Every one of those uses provenance to make the agent trust context *less*, and [13] states the cap outright: content confidence "may only lower trust within a channel's provenance ceiling, never raise it" (its v2). Outside agent memory the field is split. Credibility-aware generation multiplies attention by an estimated credibility score, so a high score amplifies a document's influence [44], and attributed question answering builds the measurement apparatus for answers that carry supporting attributions [45], though neither touches persistent agent memory and neither shows a behavioral gain from trusting attributed content more. Within agent memory we are not aware of work using provenance to make an agent trust legitimate memory *more*, and C4 exists because of that asymmetry. Bad Memory [6] sits in apparent tension with the premise, and the reconciliation is in Section 2.

Verification attached to memory. Two systems attach verification to stored memory and are the nearest neighbours to C4's intervention. VerificAgent [35] grows memory from trajectories and then applies a post-hoc human fact-checking pass, after which "the verified memory acts as a frozen safety contract" and no further updates occur; verification is a write-time human pipeline stage, and at read time the agent cannot tell a verified memory from an unverified one because everything it sees has already been filtered. SEDM [36] validates candidate memories at admission by replaying them in a self-contained execution context and admitting on a composite score; after admission the validation survives only as a scalar utility weight that reranks retrieval. Both put verification on the write path and consume it before the agent acts, so write-time curation and admission-time validation are taken, and C4 claims neither. What C4 proposes is read-time and agent-facing: a check delivered with the note that the agent can run at the moment of use and condition its behavior on. We did not locate a system that exposes that, and the Section 2 corollary bounds where it can exist at all: a claim with nothing in the workspace to check against cannot carry a checkable affordance.

Why we did not vary presentation. A large factorial study of instruction adherence in coding agent configuration files [16] manipulated four things at once: file size, instruction position, file architecture, and contradictions. Across 1,650 sessions it detected no contrast, once the many comparisons it made were accounted for in the statistics. That is absence of evidence at the sensitivity that study could reach, and we treated it as license to spend our own budget on which channel delivers the instruction and whether the agent can check it. Section 3.3 shows the license was taken too freely: our arms differ in wrapper text and note kind, so we cannot rule presentation out of any difference we report. And TriggerBench [2] establishes that acting on a latent stored

constraint when its trigger appears is substantially harder than retrieving stated facts, and degrades as context grows: the motivating failure that harness delivery exists to bypass.

8 Threats to validity and limitations

Statistical. Every cell is N=1. One seed, one agent model, two scenarios, three legs per trajectory, $17.54 total across all preserved runs ($6.34 for the reported grid). Nothing here supports an effect size, and none is computed. Two-leg rows are two dependent observations rather than two independent ones: k=3 starts from the workspace k=2 left, so a row is one event and its sequel. The cells are not independent of each other either. Within a scenario every arm's leg 2 starts from the same shared leg 1 (Section 3.1), so one idiosyncratic first session is common to all of them, and the number of independent starting draws per scenario is one. On the DISCOVERED scenario that shared ancestor is also the session that failed `annotated_tag_exists`.

Instrument. Section 4 is itself the largest threat, and it is only partly discharged. One class of scorer defect was found by reading transcripts, fixed, and re-scored at zero cost. We have no basis for believing it was the only one. The specific lesson generalizes past our bug, and Section 4.3 states it. We now compute action-derived and state-derived checks in parallel and error on disagreement, but that guard only covers facts observable both ways.

DEFECT 10, since resolved. The workspace carries forward between legs, which is deliberate, but on the TOLD scenario it means leg 2 begins with leg 1's queue entry already present. Only one thing is actually residue. `deploy/README.md` and `queue.yaml` are scenario fixtures present at leg-1 start (Section 3.2), and only the second, staging-targeted entry in `queue.yaml` is left behind by leg 1. A previous revision described both files as residue. Wrong, and corrected here. The residue has two consequences: the fact stops being uncorroborable at k >= 2, because "the deploy is already queued" is derivable from the workspace; and the primary check becomes ambiguous, because it requires the queue count to increase and an agent that correctly declines to append a duplicate fails it. Every arm failed the check for this reason.

This is no longer open. Each scenario now names the files whose carried-forward content could satisfy a later leg's check before that leg does any work. At the start of every k >= 2 leg, after the restore-integrity check and before the agent runs, the runner restores exactly those files to the content they started with, and leg-start baselines are recomputed from the reset tree. The TOLD scenario names `deploy/queue.yaml` and is the only one of the six that needed it. Its primary check now asks for at least one appended entry in every leg, since after the reset any staging entry present at leg end must have been added during that leg. All other paths keep their natural cross-leg residue. The fix is merged, and it is not retroactive: every number in the grid above comes from runs made before it. The second seed of Section 5.4 gives the first observations on the repaired surface, and its primary-check failures are behavioral rather than artifacts of this defect.

DEFECT 11, scenario-name leakage through the workspace path. The agent's working directory is named after the trajectory, so its path contains the scenario slug, and the session-init event carries that path. On the TOLD scenario the slug is `deploy_reverted_by_reconciler`, which states the fact the scenario exists to hide. The control transcript shows the agent picking the term out of its own path, searching the workspace for it, and then reasoning about reconciler behavior. The word appears nowhere in the workspace files. This independently invalidates the "uncorroborable" premise for every arm on that scenario, including the no-memory control. The DISCOVERED scenario's slug (`release_via_ci`) is also suggestive, but its transcripts show zero non-path references to it, and its fact is corroborable from the workspace by design, so we do not believe it was materially affected. Any future run must use opaque workspace directory names.

This is what [27]'s point about harness design looks like when it happens to you. This fix has since merged as well: trajectory directories are opaque run ids, the scenario slug lives only inside the trajectory state file, and leg workspaces pin a synthetic committer identity. Like the residue reset it is not retroactive, and the second seed of Section 5.4 is the first data from it.

Arms are not matched. Section 3.3: the hook and proxy arms differ in note kind and in delivered wrapper text, and the hook arm's store is contaminated with extra notes. The validity gates are not matched either. The hook arm can be invalidated by a rendering failure, since it must additionally show the note's content in the transcript, and the proxy arm has no equivalent check to fail. Legs are therefore dropped under different criteria in different arms, which is a selection asymmetry on top of the content mismatch. No channel comparison is made from this data.

The attrition itself, by arm, from the run-directory dispositions at grid close: of the ten legs then preserved outside the reported grid, control lost three to a poisoned environment (including one shared first session), proxy-plain lost four (two poisoned, two superseded by the leg-space correction), hook lost two (both superseded by the DEFECT 6 delivery fix), and verifiable lost one (superseded by the DEFECT 8 containment fix); provenance lost none. Four further would-be legs were aborted by the pre-spend probe at $0, two hook and two proxy-plain. Recorded validity failures among the ten: both poisoned proxy-plain legs (empty store at start), both superseded hook legs (injection counter zero), and the superseded verifiable leg (trail text absent from the probe's return).

Pre-registered criteria are partly unevaluable, not cleanly missed. The design pre-registered a 50% reduction in mistake rate together with a 40% reduction in turns-to-fact against control, on both headline scenarios, plus an expectation that memory arms would reach the fact within about two turns. On the DISCOVERED scenario the control's mistake rate is 0.0, so no reduction is definable. On the TOLD scenario turns-to-fact is censored by construction and, post-re-score, the mistake axis is empty. The turns expectation is missed. Our first draft overstated by how much, because the metric counts assistant events rather than conversational turns. The verdict is that most of the criterion was unevaluable in this pilot, and we would rather say that than record a clean failure the data cannot support.

Construct validity. Do the metrics measure what their names claim? Fact acquisition is scored as a verbatim match against an anticipated action, so an agent achieving the right outcome by a route we did not anticipate scores as censored. Mistake signatures cannot see intent. That is deliberate, and it means an agent that plans a mistake without executing it scores clean. Our `billable_tokens_to_fact` aggregate is defective (Section 5.1) and unused, and `turns_to_fact` is misnamed.

Internal. The note is planted by the harness and is identical within an arm. That deliberately isolates the read half of memory from the write half: whether an agent uses a note it already has, rather than whether it would have written a good one. Every number is conditional on a good note existing, and whether an agent writes a good note unprompted is unmeasured. The workspaces are synthetic and small, so no session runs long enough for its context to be compacted, and this measures survival across session boundaries only. The implementation under evaluation and the harness share an author, and Section 4 is a direct instance of the risk that creates. The mitigations that actually bit are the paired delivery gates and the preserved transcripts, which are what made the re-score possible. The others are the no-judge rule (with the qualification in Section 4.3), pre-spend aborts, retained invalidated runs, immutable originals beside re-scored records, and published artifacts.

External. One agent model, one product's hook and proxy surfaces, two of six authored scenarios, synthetic corpora. Results are directional at best.

9 Future work

The instrument comes first, and it is now the cheap part. DEFECT 10 is fixed and merged, and so are opaque workspace paths and a TOLD fixture that no longer documents the prescriptive half of its own hidden fact; Section 5.4 reports the first runs on that repaired surface. Still open:

- an anchor that can be separated from the scenario's discovery artifact;
- `anchor_checked` and `verify_command_ran` computed for every variant, so that they can support an arm contrast;
- `billable_tokens_to_fact` fixed or relabeled, with `context_tokens_at_fact` reported beside it;
- a name for `turns_to_fact` that says what it counts;
- scenario screening by control-only legs: neither scenario established the prior-contradiction premise (the control arm produced the correct action at every grid leg in both, Section 5), so candidate scenarios should first run control-only legs, at roughly $0.30 per leg, and only those with a control-verified nonzero baseline mistake rate should graduate to the grid. The single control mistake in the second seed (Section 5.4) is the first sign that the TOLD premise can establish on the repaired fixture.

One further item is not a bug at all, but a design question. A non-interactive harness scores an agent that asks permission identically to one that fails, so a scenario whose correct action is irreversible needs either a standing authorization in the prompt or a way to record the question as an outcome. HiL-Bench [43] documents the same blindness for benchmarks generally: non-interactive designs cannot distinguish an agent that guesses past a gap from one that would have asked, and builds blocker tasks around an explicit ask-human tool to make the two separable; our instance is the harsher variant, where the question was asked and there was no one to answer.

After that, the experiment: additional seeds, the remaining authored scenarios, a fourth scored leg to test whether any effect decays (the `legs/4` directories in the published run directory are the Section 5 harness-fix verification legs, not this), at least one further agent model, and arms matched on note kind and wrapper so a channel comparison becomes available. The single highest-value cell to replicate is the provenance-only variant failing the release task in both sessions, which is the only outcome-level signal in the grid. Our earlier hypothesis, that dating a note causes an agent to discount it, is not supported by the transcripts. The agent did not discount the note. It stated the fact correctly and then declined to act on grounds of irreversibility. The hypothesis worth isolating is therefore **irreversibility salience**, whether a provenance trail makes an agent treat the action it prescribes as higher-stakes and so more in need of confirmation. That is testable by varying the trail while holding both the fact and the reversibility of the required action fixed, and it wants a reversible-action scenario as its control. Second is the structural gap in Section 2: an affordance for uncorroborable claims cannot corroborate content and would have to corroborate the note's own origin chain instead.

Separately, and beyond this paper's scope, we are running a pre-registered experiment on whether a retracted belief should itself be delivered as memory rather than deleted.

10 Conclusion

The field has answered agents not using their memory by no longer asking them to use it. The harness delivers it, at the cued moment, whether or not the agent would have gone looking. We built an instrument to find out what happens next, and it told us that agents ignore correctly delivered

notes. It was wrong, in a way our own records already contradicted and no part of our pipeline compared.

An agent quoted a delivered note's wording back as its reason for not running the script, and we cannot say the note caused that, because the control reached the same place by three other routes. The provenance-only note failed the release task twice, where plain notes, verifiable notes, and no memory at all passed six of six. It failed by asking a question our non-interactive harness had no way to answer and no way to score.

We are publishing a pilot at the sample size we could afford, with a pre-registered criterion that turned out to be mostly unevaluable and is reported that way instead of as a clean miss. The claim we hold most firmly is about the instrument. We set out to ask whether an agent should act on a claim it cannot corroborate. Our instrument made a claim we could not corroborate against the transcript, and it was false. That is why we still think it is the right question to ask about memory.

Artifact availability

Published now. Scenario definitions, the mechanical scorer, the leg runner and trajectory driver, the report layer, result schemas, the re-score entrypoint, and the design document including the numbered defect record are in the Vectr repository (github.com/swapnanil/vectr) under `benchmarks/longitudinal_rediscovery/`. The prior pre-registered experiment of Section 6.2, including its frozen pre-registration, all six session transcripts, and the frozen parse readout, is published under `results/vectr-vs-bash/`; its deterministic transcript parser is at `benchmarks/vs_bash/tier1/g4_metrics.py`. The memory system under evaluation ships from the same repository.

The run directory is published beside them, under `results/longitudinal/`. Every table, quote, and count in this paper is derived from it and can be re-derived. Its layout: one directory per trajectory, named `<scenario>-<arm>-<variant>-<seed>` for runs made before the DEFECT 11 fix and `run-<16 hex>` after it, with the trajectory id inside the directory's `state.json`; each holds `legs/<k>/artifacts/` with `result.json`, the corrected `result.rescored.json` written alongside it with the original never overwritten, `transcript.jsonl`, and per-leg workspace snapshots. Leg 1 is shared and lives under `_shared/leg1/`, which also holds the scenario definition used for the run. Top-level `results.jsonl` and `results.rescored.jsonl` aggregate the trajectory legs; the shared first sessions keep their records under `_shared/leg1/`. Superseded and invalidated trajectories are retained beside the live ones under directory names carrying their invalidation reason, which is how the Section 4.1 counts can be audited.

The released copy differs from the private originals by three substitutions, disclosed in the directory's README. A real requester email address, captured into fixtures and quoted back by agents, is replaced by a synthetic identity throughout, including inside archives and the commit and tag identities of git repositories in workspace snapshots; those histories were rewritten, so commit ids inside released workspaces differ from the ids transcripts quote, and recorded content hashes verify only against the private originals. One local editor hook path and one plugin reference are redacted. Workspace git metadata directories ship renamed `_git`. A harness change pinning a synthetic committer identity in leg workspaces has since landed, but one later run still captured the real address by an agent-chosen route, so the release-time substitution remains the guarantee rather than the pin.

References

[1] S. Saha. Delivery, Not Storage: Cue-Anchored Working Memory as a Harness Property for Coding Agents. arXiv:2607.20972, 2026.

[2] TriggerBench: Investigating Prospective Memory for Large Language Models. arXiv:2606.23459, 2026.

[3] Getting Better at Working With You: Compiling User Corrections into Runtime Enforcement for Coding Agents. arXiv:2606.13174, 2026.

[4] VehicleMemBench: An Executable Benchmark for Multi-User Long-Term Memory in In-Vehicle Agents. arXiv:2603.23840, 2026.

[5] Auditing Provenance Sensitivity in LLM Agent Action Selection. arXiv:2607.20827, 2026.

[6] Bad Memory: Evaluating Prompt Injection Risks from Memory in Agentic Systems. arXiv:2607.14611, 2026.

[7] MemoryArena: Multi-Session Memory-Agent-Environment Benchmarking. arXiv:2602.16313, 2026.

[8] M. Sandelin. The First Controlled Benchmark of AI Memory in Coding Agents. Industry report, Stompy, March 2026.

[9] Ground Truth First: A Longitudinal Evaluation Instrument for Agent Memory. arXiv:2607.21962, 2026.

[10] The Last Human-Written Paper. arXiv:2604.24658, 2026.

[11] PROJECTMEM: A Local-First, Event-Sourced Memory and Judgment Layer for AI Coding Agents. arXiv:2606.12329, 2026.

[12] Portable Agent Memory. arXiv:2605.11032, 2026.

[13] Reliability-Conditional Updating for Agent Memory. arXiv:2606.22030, 2026.

[14] MemMark: Attribution Watermarking for Long-Term Agent Memory. arXiv:2605.25002, 2026.

[15] ARGUS: Influence Provenance Graphs for Agent Action Justification. arXiv:2605.03378, 2026.

[16] Instruction Adherence in Coding Agent Configuration Files. arXiv:2605.10039, 2026.

[17] E. Balli. The Test Oracle Problem in Synthetic LLM-as-Judge Corpora: Disappearance, Distortion and a Validation Protocol. arXiv:2607.13707, 2026.

[18] J. Vaghasiya, V. Bhat, M. A. Mohsin, A. Aali. Benchmarking the Benchmarks: A Validity Audit of Tool-Calling Evaluation. arXiv:2607.02577, 2026.

[19] Can Agent Benchmarks Support Their Scores? Evidence-Supported Bounds for Interactive-Agent Evaluation. arXiv:2605.10448, 2026.

[20] A. Kirgis, S. Kapoor, et al. Log Analysis Is Necessary for Credible Evaluation of AI Agents. arXiv:2605.08545, 2026.

[21] Y. Wang, F. Bianchi, et al. Automated Benchmark Auditing for AI Agents and LLMs. arXiv:2605.26079, 2026.

[22] Gilda and Gilda. Position: Evaluation Scores Are Perishable Knowledge Claims. ACL 2026. arXiv:2607.26191.

[23] Geist in the Machine. arXiv:2603.10450, 2026. (Cited for its disclosed multi-turn scoring defect.)

[24] Jiang et al. Anatomy of Agentic Memory: Taxonomy and Empirical Analysis of Evaluation and System Limitations. arXiv:2602.19320, 2026.

[25] Are Online Skill and Memory Modules Always Worth Their Tokens? A Budget-Constrained Study of Web Agents. arXiv:2606.15017, 2026.

[26] Building Effective AI Coding Agents for the Terminal. arXiv:2603.05344, 2026.

[27] J. Yi, K. Song. Measuring Harness-Induced Belief Divergence in Multi-Step LLM Agents. arXiv:2607.04528, 2026.

[28] Li, Fan, Zhuang. Auditing the Audit: Five Failure Modes in Benchmark-Validity Audits. TAIGR at ICML 2026. arXiv:2607.02586.

[29] Penfield Labs. An Audit of the LoCoMo Long-Term Memory Benchmark. Industry report, 2026.

[30] R. Xu, Z. Qi, Z. Guo, C. Wang, H. Wang, Y. Zhang, W. Xu. Knowledge Conflicts for LLMs: A Survey. EMNLP 2024. arXiv:2403.08319.

[31] Z. Su, J. Zhang, X. Qu, et al. ConflictBank: A Benchmark for Evaluating the Influence of Knowledge Conflicts in LLM. NeurIPS 2024 Datasets and Benchmarks. arXiv:2408.12076.

[32] W. Shi, X. Han, M. Lewis, Y. Tsvetkov, L. Zettlemoyer, S. Yih. Trusting Your Evidence: Hallucinate Less with Context-aware Decoding. NAACL-HLT 2024. arXiv:2305.14739.

[33] T. Zhao, J. Chen, S. Zhang, H. Zhu, Q. Lin, J. Liu. Exploring Knowledge Conflicts for Faithful LLM Reasoning: Benchmark and Method. SIGIR 2026. arXiv:2604.11209.

[34] Y. Chen, P. Qian, S. Wang, et al. Does RAG Know When Retrieval Is Wrong? Diagnosing Context Compliance under Knowledge Conflict. arXiv:2605.14473, 2026.

[35] T. Q. Nguyen, S. Desai, R. H. Anwar, et al. VerificAgent: Domain-Specific Memory Verification for Scalable Oversight of Aligned Computer-Use Agents. arXiv:2506.02539, 2025.

[36] H. Xu, J. Hu, K. Zhang, et al. SEDM: Scalable Self-Evolving Distributed Memory for Agents. arXiv:2509.09498, 2025.

[37] W. M. McKeeman. Differential Testing for Software. Digital Technical Journal 10(1), 1998, pp. 100-107.

[38] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, Z. Q. Zhou. Metamorphic Testing: A Review of Challenges and Opportunities. ACM Computing Surveys 51(1), 2018.

[39] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. IEEE Transactions on Software Engineering SE-11(12), 1985, pp. 1491-1501.

[40] S. Suwansathit, Y. Zhang, G. Gu. A Security Analysis of the OpenClaw AI Agent Framework. arXiv:2603.27517, 2026.

[41] J. Mohl, N. Gardner-Challis, M. Dubois, et al. Automated Transcript Analysis for Detecting Flaws in Agentic Benchmarks. arXiv:2607.27518, 2026.

[42] M. Chen, J. Wang, Z. Liu, Y. Wang, H. Zheng, Q. Wang. From Failed Trajectories to Reliable LLM Agents: Diagnosing and Repairing Harness Flaws. arXiv:2606.06324, 2026.

[43] T. Trinh, M. Elfeki, G. Luo, et al. HiL-Bench (Human-in-Loop Benchmark): Do Agents Know When to Ask for Help? arXiv:2604.09408, 2026.

[44] D. Adila, S. Zhang, B. Han, B. Min, Y. Wang. CrEst: Credibility Estimation for Contexts in LLMs via Weak Supervision. arXiv:2506.14912, 2025.

[45] B. Bohnet, V. Q. Tran, P. Verga, et al. Attributed Question Answering: Evaluation and Modeling for Attributed Large Language Models. arXiv:2212.08037, 2022.

[46] J. M. Rohrer, W. Tierney, E. L. Uhlmann, et al. Putting the Self in Self-Correction: Findings From the Loss-of-Confidence Project. Perspectives on Psychological Science 16(6), 2021, pp. 1255-1269. DOI 10.1177/1745691620964106.

[47] S. Heckers, H. Bauchner, A. Flanagin. Retracting, Replacing, and Correcting the Literature for Pervasive Error in Which the Results Change but the Underlying Science Is Still Reliable. JAMA Psychiatry 72(12), 2015, pp. 1170-1171. DOI 10.1001/jamapsychiatry.2015.2278.

[48] J. Kirin. Operational Proto-Introspection in Looped Language Models: Process-Quality Taps, Executable Branching, and the Readout-Control Boundary. arXiv:2607.18553 (v2), 2026.