# EVIDENCE, NOT INDEPENDENCE: REVIEWER IDENTITY ALONE DOES NOT VERIFY A CODING AGENT'S CLAIMS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

When a coding agent reports a task complete, something else must decide whether to believe it. We test whether reviewer independence, by itself, is sufficient to catch a false completion claim. In a controlled setting where a coding agent repeatedly produces a fix that passes a ticket's stated example but misses an unstated requirement, we vary who reviews (the same model; the same model reframed as reviewing a colleague; a genuinely different model), what they are told to look for, and what evidence they are shown. Reviewer identity alone never catches the omission under a neutral instruction; a stronger instruction narrows the gap for one reviewer pairing but not the other, and even then leaves a case undetected. Showing the reviewer the actual execution output resolves it in every case tested, including when the reviewer is the same agent that wrote the fix, and a visibility-matched check is consistent with the advantage coming from evidence visibility rather than reviewer architecture. A preregistered, fully-crossed replication confirms the pattern at larger scale and surfaces a cost the earlier experiments could not show: the instruction that most improves detection without evidence also raises false alarms on genuinely correct work, an effect evidence only partly offsets. A further probe finds partial evidence is not simply better than none: at an intermediate level of detail, reliability can fall below both a bare pass/fail signal and full output. An unselected random sample of real-repository tasks confirms the underlying failure is not an artifact of how examples were chosen for study. We report this as a mechanism paper, not a general solution: it establishes when execution evidence resolves the failure, leaving how an autonomous system obtains a sufficiently discriminating check without benchmark-held-out tests as open work. We translate the mechanism into a runtime control plane in which completion claims remain proposals until evidence-gated checks admit them into system state.

## 1 INTRODUCTION

When a coding agent reports that a task is complete, something else in the system must decide whether to believe it. A natural answer is to add a reviewer: a second pass, ideally by an independent party, that checks the claim before it is acted on (the standard analogy from human code review). But what does independence, by itself, actually buy?

We study this in the setting of software-engineering agents, where reviewer independence is easiest to define precisely and a worker's claim can be checked against real execution rather than taken on faith. Coding agents built on frameworks such as SWE-agent, SWE-bench, and OpenHands [2,3,11] already produce increasingly capable workers; our question is orthogonal to worker capability. It is a runtime question: once a worker has acted, what determines whether its output should be trusted, and by whom? This is an instance of the scalable oversight challenge of supervising systems that may match or exceed a supervisor's reliability [19], here in a domain with unusually well-defined ground truth. We call this the **runtime agent-management problem**.

Across a controlled pilot where a coding agent repeatedly produces a fix satisfying a ticket's literal example but failing an unstated invariant, we ask whether varying reviewer identity, instruction, or evidence, independently, is sufficient to catch the failure (formalized in Section 3).



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Detection rate for a plausible-but-incomplete fix across three probes. Each panel has its own unit and denominator; rates should not be compared across panels.

| | |
|---|---|
| **A**: Controlled pilot (12 tasks/condition; Section 5) | |
| Self-verification, reasoning only | 0/12 |
| Persona-reframed review, reasoning only | 0/12 |
| Same worker, shown execution output | 12/12 |
| **B**: Cross-model stress test (conditional subset; Sections 5–6) | |
| Neutral prompt, genuinely different model | 0/13 |
| Adversarial prompt, genuinely different model | 16/18 vs. 8/21 |
| **C**: Visibility probe (3 tasks, three architecturally distinct judges; Section 8) | |
| Discriminating evidence withheld | 1/3 |
| Discriminating evidence available | 3/3 |

The result runs against what identity-based independence would predict: reasoning-only review misses the failure under every form of independence tested: same model, a persona reframed as reviewing a colleague, or a genuinely different model under a neutral prompt (Table 1, Panels A–B), and a stronger instruction narrows the gap for one reviewer pairing but not the other. Showing the reviewer the real test output inverts this outcome completely, even when the reviewer is the worker that wrote the fix; a small matched-visibility probe is consistent with the advantage arising from possession of that signal rather than reviewer architecture, since withholding it collapses detection back down for every architecture tested, including an established coding-agent framework's own iteration loop (Table 1, Panel C).

Our contribution is primarily empirical. **(1) Reasoning-only verification under increasing reviewer separation:** self-review and a persona-reframed reviewer fail; a genuinely different model under a neutral prompt also fails, and a stronger instruction helps asymmetrically without resolving it (Sections 4–6). **(2) Execution-grounded verification and its boundary:** real execution output reliably overturns the false claim, tested further on real-repository issues and a second model (Section 7); a matched-visibility probe is consistent with evidence visibility, rather than architecture alone, accounting for the observed difference, and a finer-grained probe shows evidence and reliability relate non-monotonically (Sections 8–9). **(3) System implication:** a control plane where claims do not mutate state directly and transitions are evidence-gated, shifting the open problem from independence to verification coverage (Sections 10–11). Section 12 states precisely what this study does and does not establish.

## 2 Related work

**Coding agents and benchmarks.** ReAct interleaves reasoning with environment actions [1]; SWE-agent, SWE-bench, and OpenHands provide repository-level agent architectures and benchmarks [2,3,11]; AgentBench evaluates agents by final task success [10]. EvalPlus and HumanEval+ scale execution evidence to expose fixes passing stated examples but failing unstated edge cases [24], the narrow-fix/broad-test structure our tasks use; we add a runtime layer catching this mid-workflow. AutoGen, MetaGPT, AgentVerse, CAMEL, and ChatDev coordinate agents through conversations or roles [4–8]; once agents act, what determines whether their output is trusted is a separate question.

**Design-time optimization and self-correction.** Meta Agent Search [13], GEPA [14], AlphaEvolve [15], and Meta-Harness [16] produce stronger configurations but do not address whether an execution should be trusted. Reflexion revises strategy via self-reflection triggered by execution-grounded signals [9]; our meta-agent, not the worker evaluated, controls whether evidence is trusted. LLMs struggle correcting reasoning without external feedback [17], and self-debugging relies on execution over self-explanation [18], but neither addresses what happens when the feedback-provider's identity changes. CRITIC and LATS use execution or tool feedback within a single worker's own correction loop [22,23]; our control plane instead sits above any one worker.

**Scalable oversight and control-plane architectures.** The runtime agent-management problem instantiates the broader challenge of supervising systems that may match a supervisor's reliability [19]. Madatha's governance layer over coding-harness configuration [20] and Nikolaev's Claim Plane for parallel coding agents [21] are complementary: neither asks whether a worker's completion claim is true. What is new is not that LLM judges favor their own generations [25] or that errors correlate

across models [26], both established, but the decomposition: sufficiency tests holding two of reviewer identity, instruction, and evidence fixed while varying the third (Section 3).

## 3 EXPERIMENTAL DESIGN

**Failure structure and what we vary.** Every experiment below is built around one failure shape: a coding agent produces a fix that resolves a task's stated example but leaves an unstated invariant unaddressed, then claims completion. We construct this directly in a synthetic pilot (Section 4) and select for it among real GitHub issues via a matching criterion (Section 7): the issue's own reproduction example admits a narrower fix than the benchmark's held-out tests require. Against this failure, three things move, each held fixed while the others vary: *reviewer identity* (same model; same model told it reviews a colleague's work; a genuinely different model); *reviewer instruction* (neutral; explicit instruction to search for unhandled edge cases); and *evidence* (nothing beyond the diff; the real test output). The initial experiments below are not a full factorial design: no same-model condition is tested under the adversarial instruction, and the two adversarial directions in Section 6 review different diff sets. So they are sufficiency tests against an escalating series of candidate explanations, not an estimate of each variable's independent causal effect. Section 6.1 subsequently addresses this limitation with a preregistered crossed replication over reviewer model, instruction, and evidence, while same-writer versus different-writer effects remain confounded by diff-pool composition throughout.

**Definitions and scope.** *Execution-derived evidence* is information produced by running a check against the implementation, from a binary pass/fail outcome to full execution output (levels in Section 9). A *discriminating check* is an executed check the candidate patch fails that exercises a requirement absent from the ticket's stated example; our experiments test whether such a check overturns an otherwise plausible completion claim, not whether arbitrary execution output reveals arbitrary defects. These are sufficiency tests against a known mechanism, not a prevalence estimate (Section 12).

**Thesis.** Reviewer independence alone is not sufficient to verify a plausible completion claim; execution-derived evidence overturns it when checks expose the omitted requirement, shifting the open problem from independence to verification coverage. The sections below test an escalating series of candidates: worker self-assessment (no, Section 4); persona or model identity (no, under a neutral prompt, Section 5); a stronger reviewer instruction (helps, inconsistently, Section 6); discriminating execution evidence (yes, Section 7); and withholding that evidence (the advantage disappears, Section 8).

## 4 PLAUSIBLE COMPLETION CLAIMS CAN BE WRONG

We implemented a control-plane harness with Claude as both the planning and implementation worker, over a small synthetic Python package: sixteen single-function bugs, eight with no held-out requirement and eight built around one not stated in the ticket (non-negativity, a percentage's ceiling of 100, or a piggybacked parameter). Verification runs the task's test file independently of the worker, plus a regression check. Each task ran three times (48 runs total), retry budget two. Harness and architecture detail is in Section 10; here we report only the completion-claim failure.

**Success and recovery populations.** The eight no-held-out-requirement tasks succeeded on the first attempt in all three runs (24/24), confirming the happy path works. The eight recovery tasks are each fully fixed by the ticket's literal example but fail a held-out test the ticket never mentions. Across all 24 such runs, the worker proposed a plausible-but-incomplete fix, *claimed success*, and was contradicted by independent verification (24/24 mismatches). Re-planning on the failure evidence resolved every one, zero regressions (Appendix A.4). A zero-retry-budget probe confirmed clean escalation instead of recovery (3/3; Table 9, Appendix A.2); a concrete worked example and full per-population results are in Appendix A.4 and Table 8 (Appendix A.2).

Across all 48 runs, 0 regressions, and a claim–verification mismatch rate of 24/72 (33.3%), concentrated entirely in first attempts of the recovery tasks. The released repository contains the harness, synthetic package, all sixteen tickets, and full traces (Reproducibility Statement).

Table 2: Verification under increasing independence, same four recovery tasks and 12-run design (last row, Section 7.1).

| Configuration | Runs | Outcome |
|---|---|---|
| No verification (reasoning only) | 12 | 12/12 accepted, incorrect |
| Self-verification (same worker, reasoning only) | 12 | 12/12 accepted, incorrect |
| Persona-reframed review ("colleague" framing, reasoning only) | 12 | 12/12 accepted, incorrect |
| Same worker, real execution evidence | 12 | 12/12 correctly flagged incomplete |

Table 3: Genuine cross-model review: neutral prompt (one run per diff), adversarial prompt (three replicate runs per diff). "Consistently caught" = flagged incomplete on all three replicate reviews.

| Direction | Diffs | Consistently caught | Pooled trials |
|---|---|---|---|
| Neutral prompt, GPT-5 reviews Claude's diffs | 6 | — | 0/6 caught |
| Neutral prompt, Claude reviews GPT-5's diffs | 7 | — | 0/7 caught |
| Adversarial prompt, GPT-5 reviews Claude's diffs | 6 | 5/6 (1 mixed: 1/3 runs) | 16/18 (89%) |
| Adversarial prompt, Claude reviews GPT-5's diffs | 7 | 2/7 (4 never; 1 mixed: 2/3 runs) | 8/21 (38%) |

## 5 Is independent reasoning sufficient?

The failure in Section 4 establishes that a worker's own claim cannot be trusted. The natural next question is whether *independence* (of framing, or of the reviewing model) catches it, holding the reviewer's instruction neutral throughout this section.

**Framing- and model-level independence, neutral prompt.** We ran three verification configurations across the same four recovery tasks, 12 runs each: no verification, self-verification (same worker reasons about its own diff), and a persona-reframed condition where the same model reviews "a colleague's" change by reasoning alone. All three accepted the incomplete fix 12/12 (Table 2). The persona-reframed reviews re-derived the ticket's literal example rather than probing the held-out case (verbatim example, Appendix A.4). Framing-level independence, alone, changed nothing.

**Genuinely different model, neutral prompt.** Does a *genuinely* different model, under a neutral prompt, catch what same-model self-verification misses? We took every diff from a cross-model probe (Section 7) where the writing model's own self-verification had failed, and had the other model review it under a default, unprompted review: 13 reviews total, one run per diff (Table 3, neutral rows). All 13 claim the diff is correct, and all are wrong, with reasoning that is specific and plausible, not hedged (Appendix A.7); this is a harder, pre-filtered population than this paper's unfiltered 55.6% self-review rate, so the two are not directly comparable. Every form of independence tested here, under a neutral prompt, produced zero detections within its own set.

## 6 Stronger review instructions help, but inconsistently

Section 5 rules out one explanation for why neutral reviewers miss the failure: perhaps they simply are not asked to look hard enough. We tested this by re-reviewing the same 13 cross-model diffs under an *adversarial* prompt instructing the reviewer to search for unhandled edge cases before approving, tested only in the cross-model direction. The two directions review different diff sets (6 Claude diffs reviewed by GPT-5; 7 GPT-5 diffs reviewed by Claude), confounding reviewer model with diff difficulty. The neutral condition ran once per diff; the adversarial condition ran three times per diff (39 trials total) to rule out a single-trial artifact.

Reviewer instruction matters substantially: GPT-5 reviewing Claude's diffs consistently caught 5 of 6; Claude reviewing GPT-5's diffs consistently caught only 2 of 7, never caught 4 of the remaining 5. Pooled, this is 16/18 individual judgments (89%) versus 8/21 (38%). Replication shows the within-diff verdict is stable, but reviewer model and diff set are confounded (Section 3), so we cannot attribute the gap's size to identity alone. This rules out the objection that our neutral reviewer simply was not trying hard enough, but it does not close the gap uniformly, and even the better direction leaves a diff undetected. The pattern is bimodal at the diff level, suggesting per-diff difficulty drives the asymmetry more than within-diff reviewer variance; in cases Claude missed, its own reasoning frequently surfaces the exact defect before dismissing it as out of scope (Appendix A.7).

### 6.1 PREREGISTERED REPLICATION: DETECTION AND FALSE-ALARM RATES

The cross-model probe above selects diffs and directions after seeing which self-verifications failed. As a preregistered check fixed before any run, we crossed 10 fresh diffs (5 incorrect, 5 correct) with 2 reviewer models $\times$ 2 instructions (neutral, adversarial) $\times$ 3 evidence arms (E0 none, E1 pass/fail only, E2 full output), one call per cell (120 substantive calls, plus 24 canary calls used only as a sanity check, 24/24 correct). Nine of the 120, all in the GPT-5$\times$adversarial condition (30% of that cell), returned empty responses from the reviewer exhausting its token budget on hidden reasoning under that specific prompt; excluded per our preregistered unparseable-response rule and logged as a deviation (root cause fixed for future runs, not backfilled here). 111 calls remain.

*Detection* (an actually-incomplete diff correctly flagged) replicates this paper's central pattern at preregistered scale: with no evidence, detection is unreliable under a neutral prompt (3/10) and improves under an adversarial prompt but not to certainty (6/9, exclusion-reduced); with either evidence arm, detection reaches ceiling regardless of model or instruction (39/39). This is strong evidence that a ground-truth-linked execution signal dominates reasoning-only review, not evidence that an unaided reviewer could discover the same omission unprompted, since disclosing that execution failed is itself a strong hint the completion claim is false.

*False alarms* (an actually-correct diff wrongly flagged) show a pattern this paper had not previously isolated. Under a neutral prompt, false alarms are rare with no evidence (2/10) and absent with evidence (0/20). Under the adversarial prompt they rise sharply with no evidence (5/8, exclusion-reduced) and remain substantial even with evidence present (6/15); evidence lowers but does not eliminate the adversarial instruction's false-alarm cost, unlike its effect on detection. The instruction that most helps a reviewer catch a real defect without evidence also makes it more likely to invent one that is not there. We report this as preliminary: 5 diffs per label, further thinned by the exclusions above; a same-writer-model contrast was also examined but is confounded by pool composition (no GPT-5-authored correct diff exists to review) and is not reported as a finding.

In sum: reviewer independence alone, under a neutral prompt, is not sufficient (Section 5); a stronger prompt helps substantially in one direction and much less in the other, without becoming sufficient in either (not independently-identified causal effects, Section 3); and, newly, that same stronger prompt trades detection gains for a false-alarm cost that evidence only partly offsets (Section 6.1). The next section asks whether evidence succeeds where instruction falls short.

## 7 EXECUTION-GROUNDED VERIFICATION

### 7.1 CONTROLLED COMPARISON

Returning to the same four recovery tasks and 12-runs-each design as Section 5: does catching the false claim require independence, or specifically grounding in execution? The same worker, shown the actual failing test output, inverted the pattern completely: 12/12 correctly flagged the fix incomplete (Table 2, last row), despite never being independent of the worker that wrote the fix. Independence *alone, under neutral review* bought nothing (Section 5), while a discriminating check shown to the same non-independent worker sufficed in every studied case (repair and re-planning are each independently sufficient to resolve it once caught; Appendix A.5).

### 7.2 SELECTED REAL-REPOSITORY CASES

The controlled comparison above establishes the phenomenon on tasks we designed, not that it generalizes beyond a synthetic package built to exhibit it. As an external-validity probe, we selected six `django/django` issues from SWE-bench Verified [3] with the same structural property: four exhibited the trap once run, one did not, one produced no diff and is excluded. For each task, a fresh agent explores the real repository and produces a diff under *self-verification* and separately under *execution-aware judgment* (shown the real held-out test output).

Execution-aware judgment correctly rejected the incomplete fix in all four trap-exhibiting tasks (Table 4). Self-verification was confidently wrong on two of the four (Appendix A.6). This small probe is evidence the mechanism is not confined to code we wrote ourselves, not a rate estimate.

**Multi-repository expansion.** We extended the probe to scikit-learn, matplotlib, astropy, and sympy using a tightened two-part threshold (Appendix A.6). Of 163 Verified instances screened, 13 were

Table 4: Execution-aware judgment vs. self-verification: real-repository probe (Claude, Section 7.2) and cross-model replication (GPT-5, Section 7.3), one run per task per condition.

| Condition | Runs | Outcome | Median time / cost |
|---|---|---|---|
| Claude, real-repository probe | | | |
| Execution-aware judgment | 4 | 4/4 correct rejections | 43.8s / $0.151 |
| Self-verification | 4 | 2/4 confidently wrong; 2/4 correctly rejected[†] | 67.4s / $0.141 |
| GPT-5, cross-model replication | | | |
| Execution-aware judgment | 8 | 8/8 correct rejections | 30.6s / $0.036 |
| Self-verification | 8 | 7/8 confidently wrong; 1/8 correctly rejected[‡] | 40.2s / $0.053 |

[†]One rejection caught a genuinely incomplete fix by reasoning alone; the other diff was itself a no-op. [‡]`django-13346`, the same task Claude's self-verification also caught unaided, independently, on a different model.

selected before any diff was generated. Pooled across the 13 instances (26 diffs), self-review detected 5 of 9 actually-incorrect first patches (55.6%), execution-aware judgment 8 of 8 (100%) (Table 10). The pattern is not repository-uniform: misses concentrate in astropy and sympy, matplotlib shows no gap (Appendix A.6). This is evidence the phenomenon generalizes beyond django, not that it is repository-independent.

### 7.3 Second-model replication

Every result so far uses Claude as planner, executor, and reviewer, leaving open whether the finding is model-specific. We repeated self-verification versus execution-aware judgment on GPT-5 on the same eight tasks (four synthetic, four real), one run per task per condition.

The result replicates cleanly: execution-aware judgment caught the incomplete fix in all 8 tasks; self-verification was confidently wrong in 7 of 8, the same reasoning shape as Claude's (Appendix A.7). This small probe shows the finding is not an artifact of one model.

Across this section's probes, execution-aware judgment rejected every incorrect patch presented to it (12/12 controlled comparison; 4/4 real-task probe; 8/8 multi-repository expansion; 8/8 second-model probe), holding independence fixed or absent throughout.

### 7.4 Unselected-sample prevalence check

Sections 7.2–7.3 select real-repository instances already known to exhibit the narrow-fix/broad-test structure, which shows the mechanism generalizes but not how often it arises without that selection. Under a protocol fixed before any instance was drawn or labeled, we drew $N = 100$ instances uniformly at random from the full SWE-bench Verified population, ran one coding-agent attempt per instance, and hand-labeled two structural properties blind to any reviewer verdict: *trap-eligible* (the instance's own held-out tests admit the narrow-fix/broad-test structure, independent of what the agent did) and *trap-exhibited* (the candidate patch, when unresolved, concretely exhibits that failure, checked among the 25 unresolved-yet-patched instances in this draw). Ten of 100 instances were trap-eligible (10%, Wilson 95% CI 5.5–17.4%), and 2 of the 25 unresolved-with-patch instances concretely exhibited the trap (8%, Wilson 95% CI 2–25%).

This is an existence result, not a precise prevalence estimate: labeling was single-coder and the trap-exhibited denominator is small, so the interval is wide. Its value is what it rules out: that the failure mode studied in Sections 4–7.3 is an artifact of the selection criterion used to find real-repository examples. Both properties were observed in an unselected random sample, not only in instances chosen to contain them.

The next section asks what happens when the discriminating signal is unavailable.

## 8 Probing the role of evidence visibility

Is our verification layer adding real value, or would a standard framework's own built-in iteration already avoid the same trap? We ran SWE-agent [2], default configuration, unmodified, on the same model (Claude) and the same four trap-exhibiting real tasks from Section 7.2, one run per task, graded through the identical SWE-bench harness (per-task outcomes, Table 11, Appendix A.2).

Of the three gradable tasks, two fail in the same confident-but-incomplete shape as our own executor, for two structural reasons (Appendix A.9): held-out assertions absent from SWE-agent's local

checkout, and a test suite it never ran. The remaining task succeeds, but our own single-shot self-verification also resolved it unaided. As run, this comparison is confounded: SWE-agent structurally could not see the held-out assertions our own judgment was given directly, so a gap between the two is uninformative about architecture.

**Controlling for visibility.** To isolate the confound, we re-ran our execution-aware judgment on the same three tasks with held-out assertions withheld, leaving only what SWE-agent's local checkout could see: same diffs, only the evidence shown to the judge changed.

Table 5: SWE-agent comparison under matched visibility (same three tasks as Table 11). "No" visibility: held-out `FAIL_TO_PASS` assertions withheld.

| Condition | Held-out visibility | Runs | Outcome |
|---|---|---|---|
| SWE-agent (default config, own iteration loop) | No | 3 | 1/3 |
| Self-verification (our harness, reasoning only) | No | 3 | 1/3 |
| Execution-aware judgment, visibility withheld (new) | No | 3 | 1/3 |
| Execution-aware judgment, full visibility (original) | Yes | 3 | 3/3 |

Under matched blindness, three architecturally distinct methods (SWE-agent's own iterative loop, our harness's plain reasoning-only self-verification, and our own execution-aware judgment with the held-out evidence removed) converge on the identical 1/3, while the only condition with held-out visibility reaches 3/3. *Observed:* 3/3 with the discriminating signal present, versus 1/3 across three architectures once withheld. *Interpretation:* consistent with evidence visibility, not architectural differences, accounting for the gap. *Implication:* this revises what the original comparison could claim: not that our control-plane architecture outperforms SWE-agent's; under equal information, our own judgment does no better than either baseline. We report this because it sharpens the paper's central claim: this small probe (three tasks) is consistent with the discriminating signal being the active ingredient, not a particular architecture (Appendix A.9).

## 9 Exploratory: partial-evidence effects

The comparisons so far are binary: no evidence versus complete raw output. We constructed three intermediate levels and evaluated all five on the seven GPT-5 tasks where self-verification was genuinely wrong: Level 1 a bare pass/fail signal, Level 2 failing test names only, Level 3 the full assertion for the failing test(s), Level 4 the complete raw output. Levels 1–3 each ran three times per task (21 trials per level) to rule out a single-trial artifact; Levels 0 and 4 are anchors run once per task. We label this exploratory throughout: it rests on a small task set (Section 12).

The sequence is not monotonic. At Level 2, GPT-5's pooled rate drops to 62% (95% CI 41–79%, descriptive over three correlated runs per task), below the 100%/95% reliability at Levels 1 and 3. The dip recurs within individual tasks rather than concentrating in one trial: `ticket_08_progress` wrong on all 3 runs, `ticket_07_discounts` and `ticket_13_eligibility` each wrong on 2 of 3, `ticket_15_timeout` on 1 of 3, so GPT-5 rationalizes partial evidence on a majority of tasks, not noise confined to one outlier (Level 3 dips modestly too; Appendix A.8). Bare pass/fail outperforming test-names-only evidence shows partial evidence can be less reliable than either none or full, not merely intermediate between them.

**Replication on a second model.** We repeated the construction on Claude over 14 task-instances where its self-verification was independently confirmed wrong, scored once per level (full list, Appendix A.12). Claude is flat at 14/14 for Levels 1, 2, and 4, with one soft spot at Level 3 (13/14); the same task and level GPT-5 also dipped on it (Appendix A.8). Pooled, Levels 1–3 sit at 41/42 (≈98%, CI 88–100%) against GPT-5's sharper Level-2 dip (62%, CI 41–79%), suggesting the mechanism is task-dependent as well as evidence-dependent.

## 10 Runtime implication: claims ≠ state

Sections 4–9 establish an empirical pattern: a worker's claim of success cannot be trusted (Section 4); reviewer independence does not fix this (Section 5); discriminating execution evidence does, within the bound Section 8 identifies; and evidence can still be insufficient (Section 11). Each finding motivates a corresponding design choice: claims must not directly mutate state, transitions must be evidence-gated rather than identity-gated, recovery and escalation must be available when evidence contradicts a claim, and passing configured checks must not be equated with universal correctness.

Table 6: Evidence-quality gradient (GPT-5), seven tasks where self-verification (Level 0) was wrong. Levels 1–3 pooled across three runs/task; Levels 0, 4 are single-run anchors. CIs are descriptive.

| Level | Correct | 95% CI | What the reviewer saw |
|---|---|---|---|
| 0 (none, self-verification) | 0/7 | — | Reasoning alone |
| 1 (binary signal) | 21/21 (100%) | 85–100% | Pass/fail only |
| 2 (failing test names) | 13/21 (62%) | 41–79% | Test names, no assertion content |
| 3 (targeted failure detail) | 20/21 (95%) | 77–99% | Full assertion for the failing test(s) only |
| 4 (full suite output) | 7/7 | — | Complete raw output |

Together these motivate a runtime supervision architecture separating agent claims from operational state: the control plane admits completion claims only through evidence-gated transitions, with bounded recovery and escalation when verification fails.

### 10.1 Agent communication is not system state

A common multi-agent abstraction treats coordination primarily as message passing, but natural-language history is a weak representation of operational state: a worker reporting "the issue is fixed and all tests pass" implicitly asserts facts the message alone cannot establish, coupling reasoning with execution: exactly what Sections 4–5 show fails. We instead maintain explicit workflow state $s_t = (\text{task}, \text{repository}, \text{stage}, \text{artifacts}, \text{verification}, \text{budget})$, updated only through validated transitions: a meta-agent observes $s_t$ and evidence $o_t$ and selects $\pi_M(s_t, o_t) \to (x_{t+1}, c_{t+1})$, but the worker's response is candidate evidence $r_{t+1}$ that a validation function admits or rejects, $V(s_t, x_{t+1}, r_{t+1}) \to s_{t+1}$. A worker may *propose* completion, but verification state is produced only once required execution tools return valid evidence (Appendix A.3).

### 10.2 Architecture

Four mechanisms implement this: constrained delegation (workers get only the context and tools their stage requires), deterministic transition guards (completion requires configured checks the meta-agent cannot redefine), independent verification (separate from asking an executor to check its own work), and bounded recovery and escalation (failed verification triggers repair or re-planning, escalating once the retry budget is exhausted). Full algorithmic detail, the architecture diagram, and the ticket-to-change instantiation used for Section 4's pilot are in Appendix A.3 and Appendix A.1.

## 11 Who decides coverage?

**Coverage, not just presence, of evidence.** Section 8's visibility-matched probe isolates a sharper version of this paper's central problem: verification must not just exist, it must test the specific claim the fix needs to satisfy. Evidence *presence* is not evidence *coverage*: our architecture can establish that configured checks passed, not that those checks cover everything genuine completion requires. The gap recurs elsewhere: both of self-verification's real-repository misses (Appendix A.6) involved a fix satisfying one requirement while a second went unchecked, and the evidence gradient's Level-2 dip (Section 9) shows a reviewer given a failing test's name, but not what it asserts, filling the gap with a guess. In each case, "a check ran and passed" and "the right check ran" are conflated.

A natural response makes the control plane responsible for evidence *coverage*: an evidence-planning step decomposes the completion claim into required sub-claims, and completion requires both configured checks passing *and* sufficient coverage against that decomposition (Appendix A.10). But an LLM-backed planner decomposing from the ticket alone inherits the same reasoning-only unreliability demonstrated throughout, reproducing the failure one level higher. A useful gate instead grounds the decomposition in constraints external to the supervisor's own reasoning: mutation testing, code-coverage analysis, explicit acceptance criteria, property-based tests, static contracts. None guarantee *semantic* completeness, but each avoids a second LLM's free enumeration.

We tested the narrowest version against real data: acceptance criteria hand-authored from issue text, checked by substring match, no LLM judgment involved. Scored over all 13 real, Docker-graded attempts (Table 14, Appendix A.10), 12 failed grading outright, where coverage is moot; the one that passed was flagged `COVERAGE_GAP`, an *environment*-coverage gap no agent effort could have closed.

Determining whether verification coverage is sufficient is a nested instance of the scalable oversight problem this paper opens by invoking [19]: moving verification outside the worker resolves the failure

studied here, but the entity now doing the grading has its own boundary on what it can establish, relocatable but not eliminable (Appendix A.10).

## 12 LIMITATIONS

We report these limitations explicitly rather than let the headline numbers imply more than the data supports. This study primarily demonstrates a failure mechanism, not a precise prevalence estimate: most tasks are built around, or selected for, the narrow-fix/broad-test structure (Section 7), and a preliminary, single-coder check on an unselected random sample (Section 7.4) indicates the failure is not confined to instances selected to contain it, without supporting a precise population rate. "Independence" here means framing or model identity, not a human reviewer, except where directly tested. Adversarial instruction raises detection substantially but asymmetrically across the two reviewer pairings in the initial comparison, where reviewer model is confounded with diff set. The preregistered replication in Section 6.1 crosses reviewer model, instruction, and evidence, but does not identify a clean same-writer versus different-writer effect because writer identity remains confounded with pool composition. The preregistered false-alarm result (Section 6.1) is preliminary: 5 diffs per label, further thinned by 9 excluded GPT-5$\times$adversarial calls (a logged, non-random deviation), so its same-writer-model contrast reflects pool composition rather than a general effect. The multi-repository expansion is not repository-uniform (Table 10), and the evidence-quality gradient rests on a small task set (Section 9 rules out single-trial noise). Two leakage paths were found and removed during task construction (Appendix A.11); the verification-weighted delegation metrics ($D'$, $U'$) are structural sanity checks, detailed in Appendix A.1.

## 13 CONCLUSION

We study the runtime problem of deciding whether a coding agent's claim of task completion is true. This is a mechanism paper: it establishes when reviewer reasoning fails and when execution evidence corrects it, not a general solution to coding-agent verification. One pattern holds across every probe: reasoning without executing misses a plausible-but-incomplete fix under *neutral* review, whatever the reviewer's identity; a stronger instruction helps asymmetrically, at a false-alarm cost (Section 6.1); execution evidence reliably catches the failure when a discriminating check exists, while the matched-visibility probe is consistent with evidence visibility, rather than architecture alone, accounting for the observed advantage; and partial evidence can be less reliable than none or full (Section 9). We translate this into a control plane treating a worker's claim as a proposal, not a fact (Section 10): it resolves this failure once a discriminating check is available, without guaranteeing that such a check exists or that passing it means full coverage (Section 11), relocating, not eliminating, oversight, and leaving how an autonomous system obtains that check without benchmark-held-out tests as open work.

### ETHICS STATEMENT

Meta-agents that supervise software-engineering workers can increase the scale and autonomy of actions on code and infrastructure. Four risks follow from the design in Section 10: insufficient verification coverage (24/24 reasoning-only runs accept an incorrect fix as final); over-broad worker permissions; trace retention (ours required none, using only a purpose-built codebase and public repositories); and execution security (running untrusted, agent-generated code risks system corruption absent an isolated environment). Deployments should apply least-privilege access, repository-specific validation, mandatory human review, and secure sandboxing.

### REPRODUCIBILITY STATEMENT

The control-plane harness, synthetic repository, every experiment script, and all traces (177 main-study, 39 adversarial cross-model, 107 evidence-gradient, 26 multi-repository expansion) are publicly available.[1] Appendix A.12 maps every trace to its section and table; the $D'/U'$ scoring script and output (Appendix A.1) are in the same repository.

[1] https://github.com/iclr2027agent/meta-agent-pilot

### AI USE STATEMENT

In accordance with the ICLR 2027 policy on LLM use, we disclose using an AI assistant (Claude, Anthropic) to help design parts of the experimental methodology (e.g., the persona-reframed and adversarial-review conditions and task-selection criteria for the real-repository probes) and to draft manuscript text, format tables, and cross-check claims against the released traces. All design decisions, interpretations, and final content are the authors' own; every quantitative claim was independently verified against the released traces.

## REFERENCES

[1] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. & Cao, Y. (2023) ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

[2] Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. & Press, O. (2024) SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*.

[3] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. & Narasimhan, K. (2024) SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*.

[4] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A.H., White, R.W., Burger, D. & Wang, C. (2024) AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Conference on Language Modeling*.

[5] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S.K.S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C. & Schmidhuber, J. (2024) MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*.

[6] Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Chan, C.-M., Yu, H., Lu, Y., Hung, Y.-H., Qian, C., Qin, Y., Cong, X., Xie, R., Liu, Z., Sun, M. & Zhou, J. (2024) AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*.

[7] Li, G., Hammoud, H.A.A.K., Itani, H., Khizbullin, D. & Ghanem, B. (2023) CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*.

[8] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z. & Sun, M. (2024) ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

[9] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K. & Yao, S. (2023) Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*.

[10] Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y. & Tang, J. (2024) AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*.

[11] Wang, X., Li, B., Song, Y., Xu, F.F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H.H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H. & Neubig, G. (2024) OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

[12] Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W.X., Wei, Z. & Wen, J.-R. (2024) A survey on large language model based autonomous agents. *Frontiers of Computer Science* **18**(6):186345.

[13] Hu, S., Lu, C. & Clune, J. (2024) Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

[14] Agrawal, L.A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M.J., Jiang, M., Potts, C., Sen, K., Dimakis, A.G., Stoica, I., Klein, D., Zaharia, M. & Khattab, O. (2025) GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*.

[15] Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A.Z., Shirobokov, S., Kozlovskii, B., Ruiz, F.J.R., Mehrabian, A., Kumar, M.P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P. & Balog, M. (2025) AlphaEvolve: A coding agent for scientific and algorithmic discovery. Technical report, Google DeepMind. *arXiv preprint arXiv:2506.13131*.

[16] Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O. & Finn, C. (2026) Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*.

[17] Huang, J., Chen, X., Mishra, S., Zheng, H.S., Yu, A.W., Song, X. & Zhou, D. (2023) Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*.

[18] Chen, X., Lin, M., Schärli, N. & Zhou, D. (2023) Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

[19] Bowman, S.R., Hyun, J., Perez, E., Chen, E., Pettit, C., Heiner, S., Lukošiūtė, K. et al. (2022) Measuring progress on scalable oversight for large language models. *arXiv preprint arXiv:2211.03540*.

[20] Madatha, P. (2026) A deterministic control plane for LLM coding agents. *arXiv preprint arXiv:2606.26924*.

[21] Nikolaev, M. (2026) Claim Plane: Enforceable change intents and dynamic scope for parallel coding agents. *arXiv preprint arXiv:2607.21909*.

[22] Gou, Z., Shao, Z., Gong, Y., Shen, Y., Yang, Y., Duan, N. & Chen, W. (2024) CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*.

[23] Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H. & Wang, Y.-X. (2024) Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*.

[24] Liu, J., Xia, C.S., Wang, Y. & Zhang, L. (2023) Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models via EvalPlus. In *Advances in Neural Information Processing Systems*.

[25] Panickssery, A., Bowman, S.R. & Feng, S. (2024) LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems*.

[26] Kim, E., Garg, A., Peng, K. & Garg, N. (2025) Correlated errors in large language models. In *International Conference on Machine Learning*.

# A ADDITIONAL EXPERIMENTAL DETAIL

This appendix contains supplementary detail that supports but is not required to follow the main-text results: illustrative examples, a threshold ablation, verification-weighted delegation metrics, and full narrative detail behind the summary paragraphs in Sections 7–8.

## A.1 EVALUATING DELEGATION QUALITY AND VERIFICATION-WEIGHTED METRICS

**Case study: ticket-to-change workflow.** We instantiate the control model of Section 10 in an enterprise software-engineering workflow, task/repository resolution, planning, constrained implementation, independent verification, a completion/repair/re-plan/escalation decision, and reporting, all deterministic except where reasoning is required, pull-request publication kept behind human approval, so failures can be attributed to worker capability or to management decisions rather than the control plane itself. Section 4 reports the live pilot; its verification-weighted delegation metrics ($D'$, $U'$) are reported here rather than in the main text, since they are a structural sanity check on the recovery mechanism rather than a discriminator among the verification strategies compared in Sections 5–7.

Existing agent benchmarks such as AgentBench [10] evaluate agents primarily by final task success, which conflates worker capability with management quality: two meta-agents can use identical workers and reach the same patch via substantially different trajectories. We therefore evaluate at three levels. **Task-level** measures capture completion, build/test success, human acceptance, and appropriate escalation. **Step-level** measures evaluate management decisions themselves (delegation accuracy, recovery rate, unnecessary-action rate, invalid tool calls, coordination loops). **System-level** measures capture wall-clock time, calls, human interventions, retry depth, and cost. Three step-level metrics follow this template:

$$D = \frac{\text{appropriate agent delegations}}{\text{total agent delegations}}, R = \frac{\text{recoverable failures resolved}}{\text{recoverable failures}}, U = \frac{\text{unnecessary actions}}{\text{total agent delegations}}$$

computed over delegations to reasoning agents, since only agent choices are within the meta-agent's discretion. A delegation counts as *appropriate* if it targets a worker with the correct role and permissions for the current stage (e.g. verification is never delegated to the implementation worker), uses well-formed tool syntax, and stays within the constrained-delegation scope of Section 10 (the worker receives only the context its stage requires); *unnecessary* actions are delegations not required to reach the next valid transition (e.g. a redundant re-verification of an already-verified artifact). Both criteria are structural properties of the delegation itself, checkable independent of whether the resulting patch is correct, which is why $D$ and $U$ cannot substitute for task-level correctness (Section 12). Delegation shape is essentially always well-formed in this pilot ($D = 1.00$, $U = 0.00$ throughout), so we treat $D$ and $U$ as structural sanity checks and report step-level evidence mainly through task correctness and $R$.

**A verification-weighted variant.** $D$ and $U$ are structural: they check whether a delegation was well-formed, not whether it produced a working fix, so $D = 1.00$ throughout is a sanity check rather than a discriminator. We therefore also report $D'$, which additionally weights each delegation by its independently-observed outcome, strictly within the scored run ($D' = 1.0$ if the delegation's fix was the one ultimately verified correct within its run, $0.5$ if this attempt was wrong but a later attempt in the same run fixed the ticket, $0.0$ if wrong and the ticket was never fixed within the run), and $U'$, which flags a waste pattern $D$ cannot see: a delegation that resubmits the identical diff as the immediately preceding failed attempt despite having been given failure feedback.

$D'$ recovers the signal $D$ was missing: the full control plane's 0.83 (attempt-0 failures that the recovery loop goes on to fix, within the same run) is separated cleanly from the 0.00 floor shared by every single-shot condition, the three reasoning-only ones because the delegation was wrong and never re-attempted, and same-worker execution evidence because its own diff was correctly flagged but likewise never carried to a fix within that single-shot run. Measured at the level of individual delegations rather than task outcomes, no single-shot condition, reasoning-only or execution-aware, reaches above the 0.00 floor; only embedding execution-grounded verification inside a full recovery loop, which requires a second attempt to exist at all, moves a delegation's outcome (0.83).

$U'$ requires a delegation to reuse an unchanged plan after a failed attempt, which is structurally impossible under the main pilot's default re-plan threshold $b = 1$ (every failure triggers a fresh plan

Table 7: Verification-weighted delegation metrics across the pilot and ablation conditions. $D$ is saturated throughout (structural well-formedness only); $D'$ and $U'$ additionally weight by independently-observed outcome, strictly within the scored run. †The four single-shot conditions (no verification, self-verification, persona-reframed review, same-worker execution evidence) each terminate after one attempt, so by the stated $D'$ definition none has a second attempt in its own run to supersede the first; all four therefore score $D' = 0.00$, wrong (or, for execution evidence, correctly flagged) and never re-attempted within that run. $D'$ scores the delegation's outcome, not the review decision layered on top of it: the execution-evidence row's $D' = 0.00$ reflects that this single-shot probe's own diff was never carried to a fix within the run, not that its judgment was wrong, it is the same condition that correctly flagged the incomplete fix in 12/12 cases (Table 2, last row).

| Configuration | $n$ delegations | $D$ | $D'$ | $U'$ |
|---|---|---|---|---|
| Full control plane (pilot) | 72 | 1.00 | 0.83 | 0.00 |
| No verification† | 12 | 1.00 | 0.00 | 0.00 |
| Self-verification† | 12 | 1.00 | 0.00 | 0.00 |
| Persona-reframed review† | 12 | 1.00 | 0.00 | 0.00 |
| Same worker, execution evidence† | 12 | 1.00 | 0.00 | 0.00 |
| Repair-only, $b = 2$ | 24 | 1.00 | 0.75 | 0.00 |
| Escalation, $B = 0$ | 3 | 1.00 | 0.00 | 0.00 |

before the next attempt; Appendix A.5), so $U' = 0.00$ under that threshold is true but untested. The repair-only probe at $b = 2$ is the one condition that can exercise it: attempt 1 reuses attempt 0's plan verbatim after real failure feedback, on all four recovery tickets across three runs each (24 delegations, 12 same-plan resubmission opportunities). All 12 resolved the ticket on the very next attempt, with zero delegations resubmitting an already-failed diff unchanged: $U' = 0.00$ on 12 tested opportunities, the one condition in this pilot where the waste pattern $U'$ targets could actually have appeared, not merely a vacuous truth about a setting where it could not.

The study reported in Section 4 instantiates only **Agentic** execution, to validate that the control plane's mechanisms are exercised correctly by a real LLM backend.

### A.2 Supporting tables

Table 8: Pilot results: sixteen tasks, three runs each ($n = 48$), retry budget 2.

| Task population | Runs | Success | Notes |
|---|---|---|---|
| Single-attempt success (8 tasks) | 24 | 24/24 | 0 repairs, 0 regressions |
| Recovery via re-planning (8 tasks) | 24 | 24/24 | resolved by re-planning; 0 regressions |
| **Total** | **48** | **48/48** | Median time 17.9s; cost/success $0.0246 |

Table 9: Effect of retry budget on the discount-capping task's identical first-attempt failure.

| Retry budget | Outcome (3/3 runs) | Median time | Median cost |
|---|---|---|---|
| 2 (main pilot) | Recover via re-plan | 26.3s | $0.0325 |
| 0 (probe) | Escalate immediately | 16.7s | $0.0145 |

### A.3 Runtime control loop and architecture: full detail

Figure 1 summarizes the design that Section 10 states compactly.

**Constrained delegation.** Workers receive only the context and tools required for their responsibility. A planner may inspect code without permission to publish changes; an implementation worker may edit an isolated working tree but cannot declare its own verification successful.

**Deterministic transition guards.** Moving to verification requires an observable repository change; moving to completion requires configured validation checks. The meta-agent selects among permitted transitions but cannot redefine their preconditions.

**Independent verification.** Verification is separate from asking an executor to check its own work (compilation, tests, static analysis, diffs, semantic consistency); a worker's claim of success is therefore a proposal, not evidence.

Table 10: Multi-repository expansion, per-repository breakdown. scikit-learn produced no incorrect patches under either condition and contributes no evidence either way.

| Repository | Self-review: incorrect / detected / rate | Execution-aware: incorrect / detected / rate |
|---|---|---|
| astropy | 3 / 1 / 33.3% | 3 / 3 / 100.0% |
| matplotlib | 2 / 2 / 100.0% | 2 / 2 / 100.0% |
| scikit-learn | 0 / — / n/a | 0 / — / n/a |
| sympy | 4 / 2 / 50.0% | 3 / 3 / 100.0% |

Table 11: SWE-agent baseline, default configuration, same four tasks as Table 4, one run each. `django-12273` is excluded (infrastructure failure; Appendix A.9).

| Task | Outcome | Turns / cost |
|---|---|---|
| django-12273 | excluded (infrastructure failure) | n/a |
| django-13346 | resolved | 101 turns / $2.35 |
| django-13512 | not resolved | 33 turns / $0.33 |
| django-14140 | not resolved | 34 turns / $0.37 |

**Bounded recovery and escalation.** Failed verification triggers bounded repair or re-planning: Execute → Verify → {Complete, Repair, Re-plan, Escalate}. *Repair* re-invokes the same implementation worker with the same plan and accumulated failure evidence; *re-plan* returns to the planning worker with the failure history so the plan itself can be revised; escalation fires once the retry budget is exhausted (Algorithm 1 below; threshold detail in Appendix A.5). Routing between Repair and Re-plan is a deterministic function of the consecutive-failure count against threshold $b$, not an inferred choice.

Algorithm 1 states the control loop summarized in Section 10 precisely.

**Algorithm 1** Runtime control loop for one ticket

**Require:** ticket $\tau$, repository $\rho$, retry budget $B$, re-plan threshold $b$

$s \leftarrow \textsc{Reset}(\rho)$ ▷ deterministic: repo reset, baseline test snapshot
$\pi \leftarrow \textsc{Plan}(\tau, \rho)$ ▷ agentic: planner proposes candidate files, steps
failures $\leftarrow 0$; feedback $\leftarrow \perp$
**for** attempt $= 0, \ldots, B$ **do**
  **if** failures $\geq b$ **then**
    $\pi \leftarrow \textsc{RePlan}(\tau, \pi, \rho, \text{feedback})$ ▷ agentic: revise the plan, not just the patch
  **end if**
  $\Delta \leftarrow \textsc{Implement}(\pi, \text{feedback})$ ▷ agentic: executor edits the working tree
  **if** $\Delta = \emptyset$ **then** ▷ deterministic guard: no observable change
    feedback $\leftarrow$ "no file changes produced"; failures $\leftarrow$ failures $+ 1$; **continue**
  **end if**
  (pass, output) $\leftarrow \textsc{Verify}(\rho)$ ▷ deterministic: independent test execution
  **if** pass **then**
    regressions $\leftarrow \textsc{RegressionCheck}(\rho)$ ▷ deterministic
    **if** regressions $= \emptyset$ **then**
      **return** COMPLETE
    **else**
      failures $\leftarrow$ failures $+ 1$; feedback $\leftarrow$ regressions
    **end if**
  **else**
    failures $\leftarrow$ failures $+ 1$; feedback $\leftarrow$ output
  **end if**
**end for**
**return** ESCALATE ▷ retry budget exhausted; hand control to a human

### A.4 Concrete instance: the discount-capping task

On the discount-capping task, the first attempt in all 18 runs where it appears in this pilot (the 3 main-pilot recovery runs and 3 runs under each of the five ablation conditions) produced the identical naive fix:

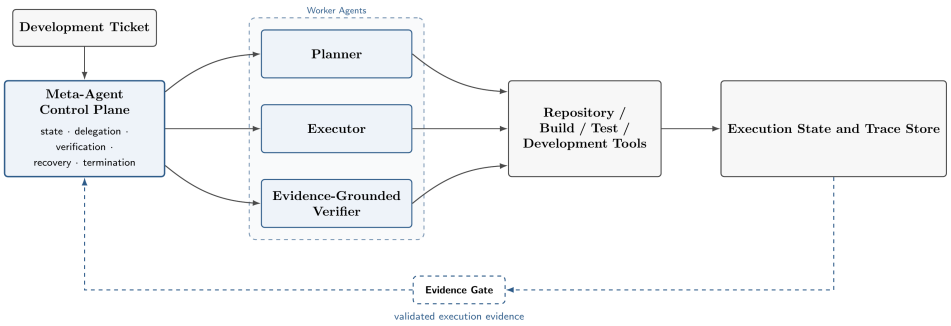


Figure 1: Runtime meta-agent supervision. Workers propose actions; tools produce observable effects; validated evidence updates state and determines whether execution continues.

```
    discount = min(discount, max_discount)
    return price - discount
```

which satisfies the ticket's literal example but fails a held-out test the ticket never mentions (`apply_discount(10, 50, 100)` returns $-40$, violating a non-negative-price invariant). Re-planning on this evidence converges, in every run, on:

```
    discount = min(discount, max_discount, price)
    return price - discount
```

which additionally bounds the discount by the price itself. Both fixes are well-formed, on-target delegations by the definition in Appendix A.1; only independent, execution-grounded verification distinguishes the wrong one from the right one. Reviewed under the persona-reframed condition, the same first fix was judged complete every time; one representative review reasoned that the change "correctly caps the discount at max_discount... which matches the expected behavior," re-deriving the ticket's own example rather than probing the price-relative case the held-out test exercises.

### A.5 Repair vs. re-plan: threshold detail

With $b = 1$ (used throughout the main pilot, Section 4), any failure immediately meets the re-plan threshold, so IMPLEMENT is never called twice under an unchanged $\pi$ with feedback in that condition, on any attempt including the last: Algorithm 1's guard depends only on failures $\geq b$, with no attempt-dependent exception. A supplementary probe using $b = 2$ instead exercises the same-plan repair path live: execution proceeds directly to IMPLEMENT without invoking REPLAN first, resolving the identical failure without re-planning; the main pilot's threshold choice, not a limitation of the architecture, is why repair does not appear in the primary 48-run results.

Re-running the same four recovery tasks with $b = 2$, giving one repair attempt before re-planning triggers, resolved the identical failure via same-plan repair alone in all 12 runs (the executor produced the corrected fix from the real test failure without the planner revising the diagnosis), at lower cost and latency than re-planning (median 17.2s / $0.0200 vs. 25.0s / $0.0315, on the original two tasks). This means an earlier framing of ours overstated the evidence: re-planning was *sufficient* for this failure, not *necessary*; both mechanisms independently resolve it, and which one fires is a property of $b$, not of the failure. The cost gap suggests a runtime manager could set $b$ adaptively, attempting repair first when a failure looks like an execution slip and escalating to re-planning when the verifier's output suggests the diagnosis itself was wrong, though we have not yet constructed a task where the initial diagnosis is wrong as distinct from one where execution alone is wrong.

### A.6 Real-repository probe: full detail

On two of the four trap-exhibiting tasks, self-verification confidently affirmed a fix that satisfied the issue's own reproduction example while missing a second, separately-located requirement the issue text never mentions: on one, the model's fix corrected a single-inheritance case shown in the ticket but not the multi-level inheritance case the tests also require; on another, the model correctly patched `JSONField.prepare_value` in `django/forms/fields.py` but missed that `django/contrib/admin/utils.py`'s `display_for_field` also needed

Table 12: Multi-repository expansion, selection funnel per repository. "Verified pool" is each repository's total instance count in SWE-bench Verified; "manually verified" is the number read against the full selection threshold after an initial heuristic pre-filter.

| Repository | Verified pool | Manually verified | Selected | Boundary | Rejected |
|---|---|---|---|---|---|
| scikit-learn | 32 | 15 | 2 | 0 | 13 |
| matplotlib | 34 | 9 | 3 | 2 | 4 |
| astropy | 22 | 6 | 4 | 0 | 2 |
| sympy | 75 | 9 | 4 | 0 | 5 |
| **Total** | **163** | **39** | **13** | **2** | **24** |

updating, since its display logic called `field.get_prep_value(value)` rather than the `json.dumps(..., ensure_ascii=False)` the ticket's Unicode-escaping fix actually requires. The model's own execution-aware judgment on this second task correctly identified the fix as incomplete and correctly named `admin/utils.py` as the missing location, verbatim: "The admin test fails because `django/contrib/admin/utils.py` also needs `ensure_ascii=False` in its `json.dumps` call for displaying JSON fields." Its account is imprecise in one narrower respect: `admin/utils.py` does not yet have a `json.dumps` call to add a keyword to at that point; the fix instead requires replacing its existing `get_prep_value(value)` call with a new `json.dumps(...)` call. The verdict was right, and grounded in real evidence, even though this detail of the model's account was not fully accurate. A third case, where reasoning alone (under self-verification, on a different run) had already caught the problem unaided, shows that reasoning-only verification was not uniformly unreliable on real tasks the way it was, by construction, on synthetic ones.

A fifth real task, selected by the same criterion, did not exhibit the trap at all: both conditions independently produced a complete, substantively different fix (a new `_resolve_output_field` method) that passed every held-out test on the first attempt. This mirrors the two boundary tasks in the synthetic pilot and is reported as a boundary case rather than forced into the trap-exhibiting set; it is excluded from Table 4, whose rows describe outcomes only on tasks where a genuine gap existed to detect.

We initially observed one sub-test on one task producing inconsistent pass/fail results across otherwise identical fixes run under the two conditions. We re-ran the underlying evaluation four additional times against the same benchmark harness, testing both textually distinct but functionally identical versions of the fix twice each; all four re-runs agreed with each other and did not reproduce the original discrepancy, indicating the initial result was very likely a one-off evaluation artifact. This re-investigation also surfaced an apparent third failing test for the same task, which on tracing back to the raw output turned out to be a benign log-concatenation artifact (a parameterized test's progress marker printed without a trailing newline, run together with the next test's name) rather than a genuine third requirement; the task's held-out gap is two locations, as originally reported.

**Multi-repository expansion: selection funnel.** Table 12 gives the full attrition from Verified pool to selected task, per repository. Every candidate considered, selected or not, is documented in the released selection log with a one-line reason (Reproducibility Statement); this includes 24 candidates read and rejected on the merits (most commonly because the gold patch's fix was a single change with no plausible narrow-vs-broad fork, so there was no real temptation to under-fix) and 2 logged as *boundary candidates*: instances where the held-out tests did check an unmentioned scenario, but the gold patch's fix was the only plausible location for a fix at all (e.g. a shared base-class method), so a narrowly-aimed fix would have covered the unmentioned scenario by construction rather than by an agent's choice to generalize. Boundary candidates were excluded before any diff was generated, not after observing an outcome.

**Multi-repository expansion: per-task outcomes.** Table 13 gives the full 13-task breakdown. Three instances (`astropy-12907`, `matplotlib-22865`, `scikit-learn-13142`) resolved cleanly under both conditions' independent diff-generation attempts and are boundary cases in the same sense as Section 7.2's single django boundary task: the diff-generation attempt happened not to exhibit the trap on this run, which does not mean the task cannot exhibit it, only that it was not observed here. Five instances (`astropy-14369`, `matplotlib-20676`, `matplotlib-24870`, `sympy-13091`, and `sympy-19783`'s self-review condition) had a genuinely incorrect diff that

Table 13: Multi-repository expansion, full per-task outcomes. "Resolved" means that condition's independently-generated diff-generation attempt actually passed the held-out tests; "correct"/"MISS"/"false alarm" describe whether that condition's judgment matched ground truth.

| Instance | Self-review diff | Self-review verdict | Exec-aware diff | Exec-aware verdict |
|---|---|---|---|---|
| astropy-12907 | resolved | complete (correct) | resolved | complete (correct) |
| astropy-13033 | incorrect | complete (**MISS**) | incorrect | incomplete (correct) |
| astropy-13977 | incorrect | complete (**MISS**) | incorrect | incomplete (correct) |
| astropy-14369 | incorrect | incomplete (correct) | incorrect | incomplete (correct) |
| matplotlib-20676 | incorrect | incomplete (correct) | incorrect | incomplete (correct) |
| matplotlib-22865 | resolved | complete (correct) | resolved | complete (correct) |
| matplotlib-24870 | incorrect | incomplete (correct) | incorrect | incomplete (correct) |
| scikit-learn-13142 | resolved | complete (correct) | resolved | complete (correct) |
| scikit-learn-25102 | resolved | incomplete (**false alarm**) | resolved | complete (correct) |
| sympy-13091 | incorrect | incomplete (correct) | incorrect | incomplete (correct) |
| sympy-16597 | incorrect | complete (**MISS**) | incorrect | incomplete (correct) |
| sympy-17630 | incorrect | complete (**MISS**) | incorrect | incomplete (correct) |
| sympy-19783 | incorrect | incomplete (correct) | resolved | complete (correct) |

self-review, reasoning alone, still caught unaided, the same pattern already reported for one of the four original django tasks (Section 7.2), now observed five further times. One instance, `scikit-learn-25102`, produced the opposite error: its diff-generation attempt actually resolved the issue correctly under both conditions, but self-review's judgment wrongly flagged it as incomplete, a false alarm rather than the overconfidence this paper otherwise studies; execution-aware judgment on the same (correct) diff recognized it as complete. The remaining four instances (`astropy-13033`, `astropy-13977`, `sympy-16597`, `sympy-17630`) reproduce the paper's central phenomenon: an incorrect diff self-review confidently judged complete, caught only once real test output was shown.

**Multi-repository expansion: cost and infrastructure.** All 26 diff-generation-plus-judgment runs (13 instances $\times$ 2 conditions) produced a submitted diff within budget; unlike the single-repository probe's `django-14238`, no instance in this expansion failed to produce a diff at all. Total cost across all 26 runs was $7.55, total wall-clock 50.5 minutes (run sequentially); per-task cost ranged from $0.037 to $1.09, the latter on `sympy-13091`'s self-verification run, whose gold-patch scope spans 21 files. One run was interrupted mid-batch by an API-key credit exhaustion and resumed after topping up; because each task's trace is written to disk immediately on completion, the resume was clean and no run was re-attempted beyond the one that had been interrupted.

### A.7 Cross-model probes: full detail

On `django-13512`, GPT-5's self-verification states "no other related code paths are required for this fix," a confident, false claim about scope structurally identical to the pattern reported for Claude on the same task. On `django-13346`, GPT-5's self-verification correctly identified its own fix as incomplete through reasoning alone, noting that the code assumed `%s`-style SQL placeholders that do not hold across the three database backends being patched. This is the same task on which Claude's self-verification also produced a genuine, unaided catch, for an unrelated reason specific to each model's own diff (Claude's missing lookup registration; GPT-5's placeholder-substitution bug).

For the genuine cross-model review probe, a representative failure: on `django-14140`, GPT-5 reviewing Claude's diff states "[the fix] maintains backward compatibility for tuple children, handles the connector case for single non-tuple children, and the negated handling... continues to work correctly," a confident, structurally detailed, and false claim, in the same shape as every confidently-wrong self-verification result reported elsewhere in this paper.

### A.8 Evidence-quality gradient: verbatim failures

At Level 2, the model's stated reasoning on `ticket_13_eligibility`: "the single failing test mentioned likely corresponds to the above-max case and should pass with this fix," a specific, wrong guess filling an information gap that Level 1's zero-detail signal never gave the model room to construct. At Level 3, on `django-13512`, the evidence explicitly showed the test still failing, assertion diff included, under the exact diff being judged, yet the model concluded "the failing tests shown expect the non-escaped output and will pass with this change," a mismatch between the

evidence and the conclusion rather than a missing-information problem; we did not probe further to determine whether this reflects the model reading a shown current failure as if it described the pre-fix state, or some other misattribution.

**Claude replication: methodology note on Level 4 for the two real tasks.** For the two real-repository tasks in the Claude replication (Section 9), the pre-existing "execution-aware judgment" trace could not be reused as Level 4, since on inspection it judged an independently-generated diff rather than the one evaluated at Levels 0–3 for that task, which would have violated the requirement that the diff being judged stay fixed across all five levels. Level 4 for `django-13512` and `django-14140` was therefore reconstructed with a fresh reviewer call using the Level-0 diff and the real, already-captured complete test output for that exact diff, methodologically identical to how Levels 1–3 were constructed for every other task, just with nothing filtered out of the evidence shown, rather than a reuse of an unrelated result. The Claude replication's one dip (`django-13512`, Level 3; see Section 9) sits between two correctly-judged levels constructed this same way, so this deviation does not affect the reported dip itself, only how Level 4 was sourced for these two tasks specifically. The full eligibility check and per-task-instance provenance for all 14 task-instances used in this replication are in `eligible_tasks_and_evidence_levels.json` (Reproducibility Statement).

### A.9 SWE-AGENT BASELINE: FULL DETAIL

`django-12273` failed to complete across three separate attempts: SWE-agent's default, non-streaming API call hung with no response, error, or retry logged for minutes, each attempt going deeper into a growing context (steps 50, 50, and 69) before we stopped it, at a combined cost of $3.77 with no submission. This is an infrastructure limitation of the default configuration under our conditions, not a result about SWE-agent's reasoning or verification behavior, and we exclude it from the comparison entirely.

On `django-14140`, SWE-agent's own trajectory shows it running the existing test suite repeatedly, ending with all 796 local tests passing, yet the real grading harness still fails it on the two held-out assertions the ticket's fix needed to satisfy. The reason is structural: SWE-bench's held-out test additions are applied to the repository only by the grading harness's own `test_patch`, never to the agent's local working checkout, so no amount of local self-testing could have surfaced these specific assertions. On `django-13512`, SWE-agent ran the forms and JSON-field test suites repeatedly and confirmed they passed, but never ran the `admin_utils` suite at all, where the held-out requirement lives. Because SWE-agent's default configuration has no mechanism to run held-out assertions it cannot see in its local checkout, this baseline as originally run only shows that default-configuration local testing does not substitute for held-out-test visibility; Section 8's visibility-matched probe (methodology below) supplies the matched architectural comparison this initial run lacked.

The remaining task, `django-13346`, succeeds: SWE-agent's own reproduction script shows the underlying SQL substitution bug directly, and the real grading harness independently confirms both held-out tests pass with no regressions. This is the same task on which our own single-shot self-verification, with no iteration at all, also succeeded, so this success cannot be cleanly attributed to SWE-agent's own iteration budget as opposed to this specific task simply being tractable for reasoning alone. SWE-agent's default configuration places no fixed cap on turns, only a per-instance cost limit ($3 in this run), a genuine structural advantage over our single-shot executor that a single run per task cannot control for.

**Visibility-matched probe: methodology and per-task detail.** The visibility-matched condition (Section 8, Table 5) reused the three gradable tasks' existing diffs and problem statements unchanged, no new diffs were generated and no SWE-bench container was re-run, and re-issued the execution-aware judgment call with a synthetic local-test-runner transcript containing only the `PASS_TO_PASS` regression result, with every `FAIL_TO_PASS` assertion removed, simulating exactly what a local checkout without the grading harness's `test_patch` would show. `django-13346` was correctly judged complete (matching both SWE-agent's own resolution and our original self-verification's independent catch on this task). `django-13512` and `django-14140` were both judged complete, incorrectly: in both cases the judge's stated reasoning cited the passing local regression suite as evidence of completion, the identical failure shape as SWE-agent's own trajectory logs on these

same two tasks (a clean local test run mistaken for a complete fix) and as plain self-verification's reasoning-only misses reported earlier in this appendix.

This visibility-matched probe bounds only the verification step in isolation; it says nothing about the control plane's other mechanisms, which SWE-agent has no equivalent of to compare against and which are instead demonstrated separately, constrained delegation and bounded repair-vs-replan in the main pilot ($R = 1.00$; Sections 10–4) and clean escalation under a zero retry budget (Table 9). With three tasks and one baseline, this does not establish that architecture never matters, only that it did not measurably matter here once information was equalized.

### A.10 Coverage decomposition, the coverage-gate probe, and deployment discussion: full detail

**Runtime supervision complements agent optimization.** Automated harness design and self-improvement [13–16] produce stronger agent configurations, but even a strong one can encounter ambiguous state or insufficient evidence at deployment; our protocol tests whether a design-time-optimized worker remains supervisable at runtime, not only more capable. A worker's claim that it succeeded is a proposal, not evidence: our verification is external to the worker being judged, whereas Reflexion's self-reflection loop [9] and tool-integrated self-correction frameworks such as CRITIC [22] and LATS [23] keep the correction decision inside the acting agent itself, consistent with reported limits of intrinsic self-correction [17,18]. Escalation can itself be successful management: at zero retry budget (Table 9), it was cleaner and roughly half the cost of re-planning to the same result; retry budget is a deployment-relevant control parameter, not an implementation detail.

**Deployment cost of execution-grounding.** Execution-grounded verification is not free relative to reasoning-only review: it requires infrastructure to reset and maintain a deterministic repository state between attempts, and it adds the wall-clock latency of an actual build and test run on every verification pass, on top of the operational-security burden discussed in the Ethics Statement. We do not measure this latency overhead directly, since our reasoning-only conditions were run against the same harness rather than deployed without one; a deployment weighing this trade-off should budget for both the sandboxing infrastructure and the added per-attempt execution time relative to a reasoning-only reviewer.

**Sub-claim decomposition, formalized.** The deterministic transition guard of Section 10 would require both configured checks passing *and* sufficient coverage against the sub-claim decomposition described in Section 11, rather than a pass/fail gate alone:

$$\textsc{Complete} \iff \textit{required_checks_pass} \wedge \textit{evidence_coverage_sufficient},$$

distinguishing a state where `verification.status = PASSED` from one where `verification.coverage = INCOMPLETE`, so a passing but under-covered verification cannot silently transition to completion. We flag, rather than resolve, the obvious difficulty this raises, discussed in Section 11: an LLM-backed decomposition inherits the same reasoning-only unreliability this paper studies, so a coverage gate is only as useful as the external constraints (mutation testing, acceptance criteria, property tests, static contracts) it is grounded in.

**A minimal, mechanically-grounded gate against real data.** We implemented the narrowest version of this: acceptance criteria hand-authored by the authors from each real task's GitHub issue text (never inferred by a model), checked by substring match against the executed test node IDs a real SWE-bench Docker evaluation actually ran, with no LLM judgment anywhere in the coverage decision itself. We scored all 13 real, Docker-graded evaluation attempts already produced against the three SWE-agent-subset tasks (Section 8) under this gate: the original self-verification and execution-aware-judgment runs, the SWE-agent baseline's own submission, and four earlier re-evaluations of `django-13512` from the flaky-sub-test investigation in Appendix A.6 (Table 14). Twelve of the thirteen attempts failed real grading outright (`gate_decision = FAIL`), where coverage is moot; the gate agrees with, and adds nothing beyond, the existing pass/fail check. The thirteenth is the only attempt across this entire real-task subset that actually passed real grading: the SWE-agent baseline's own successful `django-13346` resolution (Table 11). The gate flagged it `COVERAGE_GAP`, not `COMPLETE`: the issue thread names an Oracle-backend-specific variant of the bug, but this instance's grading environment runs Django's suite under `-settings=test_sqlite`, so no test for that variant exists anywhere in the instance's declared test set, independent of what any agent does.

Table 14: Mechanical coverage-gate scoring over all 13 real, Docker-graded evaluation attempts on the three SWE-agent-subset tasks. Coverage is only ever discriminating on a `verify_pass = True` attempt; all 12 failing attempts short-circuit to `FAIL` regardless of coverage.

| Gate decision | Attempts | Real grading result |
|---|---|---|
| `FAIL` | 12 / 13 | all `verify_pass = False` |
| `COVERAGE_GAP` | 1 / 13 | the one `verify_pass = True` attempt (`django-13346`, SWE-agent baseline) |
| `COMPLETE` | 0 / 13 | — |

This is a different failure mode than the one motivating the gate's design, and worth stating precisely rather than folding into the same story. The synthetic pilot's coverage gaps (Section 4) are failures of *agent* behavior: a test that could have been run, and would have caught the bug, simply was not written into the fix. This real instance is a failure of *environment* coverage instead: a criterion a human correctly declared has no mechanically-checkable test available anywhere in the deployed evaluation environment, so no amount of agent effort could have satisfied it here. Even a coverage gate grounded entirely outside LLM reasoning, exactly as prescribed above, still inherits whatever gaps exist in the execution environment it checks against: a boundary condition the design as originally stated did not anticipate. This sharpens rather than resolves the open question of what a `COVERAGE_GAP` decision should trigger: our one real instance requires a case split Algorithm 1 currently has no place for: a criterion that is untestable in this environment (which no worker action can resolve, and which should escalate as a documented limitation) versus a criterion that is testable but has no test written yet (where a test-authoring delegate, Section 12's open design question, could plausibly help). One real data point cannot estimate how often each case arises; that would need either scaling this probe up or adding environment metadata to the criteria schema so the gate can distinguish the two automatically, rather than an author reading issue text and Docker eval scripts by hand, which is what produced this result. We did not integrate the gate into the live control loop or re-run the recovery/escalation mechanisms under it; this is a post-hoc scoring pass over already-real evaluation data confirming the detection mechanism behaves as designed, not a claim about downstream architectural behavior.

We raise all of this because Section 8's visibility-matched result specifically rules out "run more, or any, execution evidence" as a sufficient fix and points instead at coverage of the *right* evidence as the more precise target for future runtime control-plane work; the real-data probe above shows that target has more structure to it than presence-versus-absence of agent effort, coverage can fail at the planning level, the agent-behavior level, or the environment level, and a control plane would need to tell these apart to respond to each correctly.

### A.11 Task construction and leakage

Constructing a fair test of recovery proved harder than expected, in more than one way. Our first attempt leaked the boundary condition through the target function's own docstring; all three runs succeeded on the first attempt despite our intent, until we removed the leaked sentence. Two further leak categories surfaced constructing tasks 11–16: naming both bounds as sibling function-signature parameters (e.g. `clamp(x, min_x, max_x)`) leaks the requirement as effectively as a docstring; our fix for this, a hardcoded domain constant, introduced an undiscoverable magic number testing recall rather than inference, and was discarded before any pilot run. The criterion we converged on: a held-out requirement is fair only if derivable from the quantity's *definition*, non-negativity, a percentage's ceiling of 100, or a piggybacked existing parameter, never an arbitrary constant. All revisions were made by inspecting code for leaks before any run, never after observing a live outcome.

### A.12 Reproducibility: trace-to-result mapping

For the synthetic pilot (Section 4): all sixteen ticket specifications in full (problem statement, associated test file, and held-out assertions), and all 111 execution traces from the main pilot, escalation probe, and five ablations, each trace containing the full prompt context, the model's response, the diff produced, the independent verifier's real output, and the final claim/verification outcome. For the real-repository probe (Section 7.2): the specific SWE-bench Verified instance IDs used (`django-12273`, `django-13346`, `django-13512`, and `django-14140` for the four trap-exhibiting tasks; `django-13128` for the boundary task; and `django-14238`, excluded for repeatedly failing to produce a diff), the exact diffs produced under each condition, and the real held-out test output each decision was based on; plus, for the multi-repository expansion, the full

selection log (every candidate considered across scikit-learn, matplotlib, astropy, and sympy, selected or not, with its one-line reasoning), and all 26 diff-generation-plus-judgment traces for the 13 selected instances. For the cross-model probe (Section 7.3): all 16 GPT-5 traces, on the same four synthetic and four real tasks, in the identical format. For the evidence-quality gradient (Section 9): all 63 GPT-5 traces (seven tasks, three intermediate levels each, three independent runs per task-level cell), constructed by filtering the raw test output already released for the corresponding Level 0 and Level 4 traces rather than from any new task execution; plus 44 Claude-replication traces (14 eligible task-instances at Levels 1–3, one run per task-instance per level, plus 2 reconstructed Level-4 calls for the two real tasks; Level 0 and the remaining 12 Level-4 results are reused unchanged from existing pilot traces), 107 gradient traces combined across both models, with the full per-task-instance provenance in `eligible_tasks_and_evidence_levels.json`. For the SWE-agent baseline (Section 8): SWE-agent's own final submitted diff for each task (or, for `django-12273`, the record of the three failed attempts), its full trajectory log, and the corresponding grading harness output; plus 3 visibility-matched execution-aware-judgment traces (one per gradable task), each reusing the task's existing diff and problem statement with the held-out assertions withheld from the evidence shown to the judge, per the methodology note in Appendix A.9. For the real-data coverage-gate probe (Section 11): the hand-authored acceptance-criteria file for each of the three tasks (with an `_excluded` field documenting any official `FAIL_TO_PASS` test we judged unrelated to the declared criteria, and why), the mechanical scorer's code, and its output over all 13 real Docker-graded evaluation attempts, these attempts are not new executions but a rescoring of `report.json` output already produced for self-verification, execution-aware judgment, the SWE-agent baseline, and the four `django-13512` re-evaluation runs from the flaky-sub-test investigation in Appendix A.6, each traceable to its original real `swebench.harness.run_evaluation` Docker run. For the genuine cross-model review probe (Sections 5–6): all 13 neutral-prompt traces, each recording which model wrote the diff, which model reviewed it, and the reviewer's full reasoning, sourced from diffs already released as part of the cross-model probe; and 39 additional adversarial-prompt traces (13 diffs $\times$ 3 independent replicate runs each, run to check the neutral/adversarial divergence against single-trial noise) over the identical 13 diffs. In total, 177 execution traces are released across the main study, plus the 39 adversarial-review replicate traces, the Claude and GPT-5 evidence-gradient replication traces, the 26 multi-repository expansion traces (plus the full selection log), and the visibility-matched and coverage-gate probes described above, all added afterward.

All results come from two underlying models: Claude, used as both planner and executor throughout, and GPT-5, used for the cross-model probe under the same two verification conditions on the same eight tasks. We report exact model responses as observed at the time these experiments were run and do not claim the same prompts would produce identical outputs from a different model, a different version of either model, or the same models at a later date, which is precisely the kind of dependency our released traces let a reader audit directly.