A TRACE-BASED STUDY OF HETEROGENEOUS LLM REVIEW AND REPAIR IN A TEN-WEEK DEPLOYMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

A common safeguard for software-engineering agents is to have a second model review the first model's work. Existing measurements of this practice, however, come largely from benchmarks and simulated red-teaming. We study it in a production deployment using ten weeks of event logs from KISS Sorcar, where an Anthropic model implemented changes in a working repository and an OpenAI model reviewed them read-only under a budget cap. The corpus contains 486 tasks and 1.1M events, including tool calls, model switches, and per-step costs.

The logs show that reviewer access rarely resulted in writes: among 2,164 reviewer-attributable calls, only 3 (0.14%) performed a write, and none modified tracked source files. On the 42 tasks for which review spending can be isolated, review accounted for 9.3% of total task cost. Task narratives contain 1,943 reported reviewer findings and 1,779 corresponding fixes, but these counts are self-reported by the builder; an independent re-coding finds roughly one-fifth fewer findings. In a 60-task re-coded sample, both readers identify 5 tasks where review exposed a problem that the builder's tests had not caught, while the second reader identifies 7. Three case studies from the full corpus illustrate such findings in detail, including a code-execution path from model output and a repair that introduced a linearizability race.

These observations characterize how model-based review is used in production and what it costs, but they do not establish its causal benefit because the deployment has no comparison arm. We release the trace corpus and analysis scripts to support further study.

1 INTRODUCTION

Software-engineering agents built on frontier language models now modify production repositories, prepare releases, and are increasingly expected to verify their own work. When the same model writes a change and evaluates it, verification can inherit the model's blind spots: tests may encode the buggy behavior, self-correction can stall without an external signal, and models may favor their own outputs (Huang et al., 2024; Panickssery et al., 2024; Goel et al., 2025). A common response is heterogeneous LLM review, in which a second model, ideally from a different vendor, reviews the first model's work (McAleese et al., 2024; Xiang et al., 2026).

How effective and costly this practice is in production remains unclear. Controlled benchmark experiments find that cross-model review can help in one pairing direction and hurt in the other, the negative direction being the one our deployment uses (Xiang et al., 2026; Olausson et al., 2024); AI-control studies under simulated attacks likewise disagree on whether vendor diversity improves oversight (Greenblatt et al., 2024; Pinto, 2026; Jha, 2026). Observational studies of public pull requests measure comments and merge outcomes but not complete execution traces or costs (Li et al., 2025; Lin et al., 2026; Selvanayagam & Ghaleb, 2026), and the closest deployment report, Kim (2026), gives aggregate outcomes rather than reviewer actions and cost. Existing evidence thus leaves open a basic empirical question: what does cross-vendor review actually do in a production coding workflow, and what does it cost?

We answer this question with a post-mortem analysis of a ten-week deployment. A single maintainer, an author of this paper, used a fixed protocol in every task in a working software repository: `claude-fable-5` (Anthropic) handled development, while `gpt-5.6-sol` (OpenAI) per-

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.

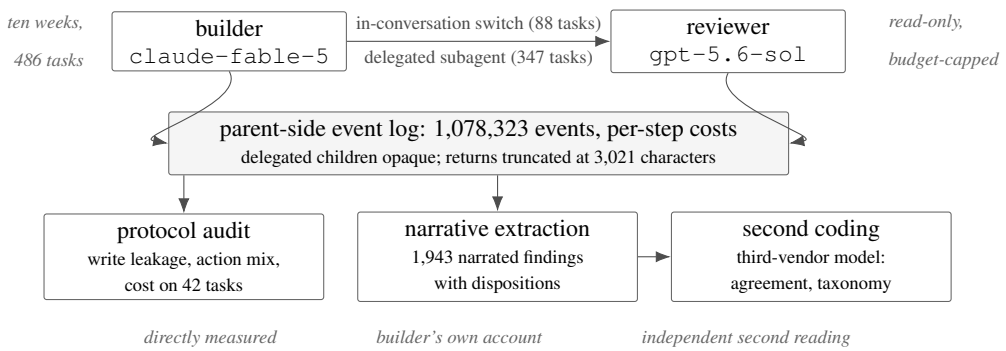

Figure 1: The deployment, its record, and the analyses. One model builds, a second-vendor model reviews through two dispatch modes, and the log is complete for the orchestrating agent and opaque below it. The analyses split by evidence level.

formed a read-only review and debugging pass under an explicit budget cap (Figure 1). The deployment used the KISS Sorcar framework (Sen, 2026), as part of the maintainer's ordinary workflow rather than a design for this study. The trace corpus contains 486 tasks and 1,078,323 events, including tool calls, model switches, streamed thoughts, and per-step costs for the orchestrating agent; the reviewer participated in 431 tasks, and the repository produced 48 tagged releases during the study period (Section 3.2).

The traces provide three views of the review process. First, they let us audit whether the reviewer followed its read-only role: of 2,164 tool calls in reviewer windows, only three (0.14%) performed a file write, all to the reviewer's own notes, against a 22% write share in builder windows, and half of the reviewer's shell commands were search audits while one in eight executed probe scripts or tests. Second, they let us measure review cost alongside reported output. On the 42 tasks for which review spending can be separated, review accounted for 9.3% of task cost; the narratives report at least one substantiated finding in 80% of 432 reviewed tasks, comprising 1,943 findings and 1,779 reported fixes. These counts are builder self-reports rather than verified repairs, and a second coding of a sample finds roughly one-fifth fewer findings.

Third, closer examination reveals cases in which review identified problems that the builder's checks had not caught. A broad verification-gap label marks 34% of reviewed tasks, but its inter-coder agreement is low ($\kappa = 0.20$), so we focus on the coded examples: in a 60-task re-coded sample, both coders identify 5 tasks in which review exposed a problem the builder's tests had not caught, the second coder 7, four of them defects demonstrated despite a green suite. Three full-corpus cases illustrate the mechanisms: a coverage gate that excluded the changed line, a code-execution path reachable from model output, and a linearizability race introduced by a repair (Appendix F). Review also continued to find problems after earlier repairs: among multi-round tasks with a determinate outcome, 60% contained a new finding in a later round.

These observations are measurements of one production deployment, not a causal evaluation of cross-vendor review. With no comparison arm, the study cannot determine whether the reviewer improved outcomes relative to no review, a same-vendor reviewer, or a reversed assignment, nor how often such review would expose defects elsewhere. We release the trace corpus and analysis scripts so that future controlled studies can test these questions directly.

Our contributions are as follows.

1. An event-level trace corpus of a cross-vendor builder-reviewer coding protocol in production: 486 tasks, over one million events, per-step cost telemetry for 299 tasks, and extracted counts of findings, reported fixes, rejections, and deferrals.
2. Measurements of review economics and behavior: review consumed about 9% of task cost where spending can be separated, the reviewer rarely wrote despite operating through tools, and later rounds continued to produce findings after earlier repairs.
3. A re-coded analysis of reviewer complementarity: 5 tasks on which both coders agree that review exposed a defect missed by the builder's checks (7 for the second coder), with three detailed cases showing how such failures arose.

4. An empirical basis for controlled studies of agent oversight: role-tagged traces, review-to-finding attribution, repair re-execution, explicit cost accounting, and comparison arms absent from this deployment.

2 RELATED WORK

Appendix D discusses each line of work in greater detail. Multi-agent coding systems assign complementary roles such as builder, reviewer, and tester (Hong et al., 2024; Qian et al., 2024; Li et al., 2023; Huang et al., 2023; Pan et al., 2025b), and a related line studies agents that critique and revise their own work (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024; McAleese et al., 2024). Limited self-correction (Huang et al., 2024; Olausson et al., 2024), self-preference (Panickssery et al., 2024), and correlated errors across models (Goel et al., 2025) motivate a reviewer from a different vendor. AI-control work studies an untrusted builder monitored by a separate model under a limited budget (Greenblatt et al., 2024); recent results disagree on how much vendor diversity reduces shared errors (Pinto, 2026; Jha, 2026).

Controlled and deployment evidence. The closest controlled study is Xiang et al. (2026). On 116 LiveCodeBench problems, Claude reviewing Codex drafts raised the pass rate from 71.6% to 89.7%, while Codex reviewing Claude drafts lowered it from 91.4% to 82.8%. The latter pairing matches our deployment, but the settings differ: their reviewer could not execute code and its revision was the final answer, whereas ours runs probe scripts and tests and returns findings the builder may reject or defer (Section 4.3), and their tasks carry a hidden correctness oracle where ours have none beyond the builder's checks. With neither a same-vendor nor a swapped-role control, our deployment cannot confirm or overturn their result.

The closest production study is Kim (2026), which ran a heterogeneous builder-reviewer pair for 27 days and reports that heterogeneous pairs flagged a defect or requested a change in 69.8% of reviews, compared with 53.1% for homogeneous pairs. Unlike our study, however, it does not provide event-level execution traces or review costs. Our study therefore complements these controlled and aggregate deployment results with the underlying actions and costs of a single deployed system.

Observational studies and agent traces. Several studies examine LLM review in real projects through review comments, pull requests, and merge outcomes (Frömmgen et al., 2024; Adams et al., 2026; Cihan et al., 2025; Lin et al., 2026; Selvanayagam & Ghaleb, 2026); Chowdhury et al. (2026), for example, report merge rates of 45.2% for agent-only review of AIDev pull requests versus 68.4% for human-only review. Such outcomes capture the final record but not the trajectory that produced it, and Karakaya et al. (2026) show that labels such as "fixed" and "rejected" are not necessarily reliable ground truth. Other work releases agent trajectories and deployment traces (Pan et al., 2025a; Li et al., 2025; Helwig, 2026; Peng et al., 2026; Richards et al., 2026). Our corpus instead follows a cross-vendor builder-reviewer protocol through the orchestrating agent's event log, exposing individual review actions and their costs.

Measurement and evidence quality. A complementary methodological literature asks whether the available measurements support the conclusions drawn from them (Salaudeen et al., 2025; Zhu et al., 2025; Kapoor et al., 2024; Cemri et al., 2025; Chillarege et al., 1992; Le et al., 2019). We follow it by distinguishing logged behavior from builder-reported outcomes, re-coding a sample with a third-vendor reader, and making no causal claim that vendor diversity improves review; the goal is to characterize what a deployed cross-vendor reviewer does, what it costs, and which evidence a controlled comparison would need (Section 5).

3 METHOD

3.1 DEPLOYMENT AND PROTOCOL

The traces come from a ten-week deployment by one maintainer, July 15 to September 21, 2026 (Pacific time), on the KISS Sorcar framework (Sen, 2026). Every task prompt carried the same review instruction: `claude-fable-5` performed the development work, while `gpt-5.6-sol` reviewed and debugged the result without modifying the main code, limited to a fixed fraction of the

Table 1: Corpus selection (top; Section 3.2, `scripts/corpus_selection.py`) and headline counts with and without the 18 study-related tasks (bottom). One kept task still running at the backup logged 194 more events afterwards (440,932 in the released database). Event-derived rows are recomputed from the log; the finding total uses the extraction table (121 findings in the 18 tasks), the disposition rows a re-reading of the 18 narratives under the rules of Section 3.3 (112 findings, 111 reported fixes, one deferral, no rejection).

	Tasks	Events
September 8 backup (July 9 to September 8)	477	659,590
kept: prompt contains `Use 'claude-fable-5`	329	440,738
dropped, top-level tasks (54 before July 16)	63	61,888
dropped, subagent records (63 reviewer runs; 61 under kept tasks)	85	156,964
Not in the backup (September 8 to 21)	157	637,391
	All tasks	**Without the 18**
Tasks	486	468
Events	1,078,323	1,044,316
Reviewed tasks	431	415
in-conversation (`set_model`)	88	85
delegated	347	334
both	4	4
Unreviewed tasks (no steps / stopped / no review)	55 (24 / 8 / 23)	53 (23 / 8 / 22)
Reviewer tool calls in in-conversation reviews	2,164	2,130
of which write-tool calls	3 (0.14%)	3 (0.14%)
Recorded spend	$16,775	$15,936
Tokens	10.5B	10.0B
Reviewer cost share (Section 4.2)	42 tasks, 9.3%	39 tasks, 9.2%
Narrated findings	1,943	1,822
Reported fixes	1,779	1,668
Rejected claims	19	19
Deferred or excluded items	108	107

task budget (a 20% limit in 168 tasks and 50% in 316, overlapping for about two and a half weeks in August; the two database-checking tasks below had no stated limit).

The tasks were ordinary development work in the maintainer's public repository, covering an agent framework (IDE extension, web application, server daemon, and public API), storage-engine and compression optimization, and technical writing. The repository contains about 690K lines of tracked source across 2,390 files, mostly Python and JavaScript with a 25K-line C++ storage engine, and the maintainer tagged 48 releases during the study window, so the reviewed code was part of an actively maintained repository.

The reviewer entered tasks through two mechanisms, both recorded explicitly in the event log: in 88 tasks the main agent switched to the reviewer model with `set_model` and later back, and in 347 it launched reviewer subagents with `run_parallel` or `run_agent` (123 with two or more reviewers; four tasks used both mechanisms). The remaining 55 tasks had no reviewer: 24 recorded no steps, eight were stopped by the user, and 23 completed without a review.

Study-related tasks. Eighteen tasks concern this study or the maintainer's own papers, flagged by `scripts/study_task_exclusion.py` from prompts mentioning the trace extract, a manuscript, or a paper. Ten built or checked the trace extract between September 8 and 21 under the same protocol, seven edited the maintainer's papers about the framework, and one is a false positive of the screen. These tasks hold 121 of the 1,943 narrated findings, at a higher per-task rate than the rest; the largest contains 67 findings across three review rounds while updating the maintainer's paper.

We retain these tasks but report headline counts both with and without them (Table 1). Removing them changes event-derived counts by 1.6% to 5% and narrative counts by about 6%; the three reviewer writes remain, and three of the 42 cost-analysis tasks are among the 18 ($6.43 of recorded cost). This makes the effect of study-related work explicit without an arbitrary exclusion from the released corpus.

3.2 TRACE CORPUS

The framework stores its records in a SQLite database with two tables. `task_history` contains one row per task with the prompt, final result, starting model, cost, token count, and step count. `events` contains 1,078,323 sequenced JSON records, including tool calls and arguments, tool results, streamed model output, system prompts, and periodic `usage_info` records with running cost and step counts.

Per-task cost totals are recorded for 458 of the 486 tasks (the remaining 28 record zero) and sum to $16,775; per-step `usage_info` telemetry begins on August 15 and covers 299 tasks. Model switches are explicit in the event stream, so each parent-side tool call can be assigned to a model, and since costs are running totals, an interval's cost is the difference between its boundary totals.

Corpus selection. The released database is itself a selection. At 19:12 on September 8, the maintainer backed up the task database (477 tasks and 659,590 events from July 9 onward), then removed every task whose prompt lacked the string `Use 'claude-fable-5`, along with its events. This kept 329 tasks and dropped 148: 63 top-level tasks (54 predating July 16 UTC) and 85 subagent records, including 63 delegated review runs. The database was then extended with 157 tasks through September 21. The backup was retained, including the dropped child events and cost totals; Appendix C.1 audits both using `scripts/backup_subagent_audit.py`.

3.3 FINDING EXTRACTION AND DISPOSITIONS

Review findings and their outcomes are described in each task's final result record. We use a language model, `claude-fable-5`, to convert these narratives into a structured table using a released extraction prompt recording, per task: review rounds; confirmed findings; findings reported fixed, rejected as false alarms, or left unfixed with a stated reason; whether the final review was clean; the highest severity term; whether a real finding appeared while the builder's tests were passing; findings first reported in round two or later; and whether the review rejected an entire design.

We use conservative counting rules. When a narrative gives an explicit number, we use that number; when it gives only a vague description, the value is treated as unknown and omitted from the totals. A finding counts as confirmed only when the narrative says that it was confirmed; unchecked suggestions do not count. *Repaired* means that the narrative reports the problem as fixed; we do not re-run the repair. *Rejected* means that the narrative provides evidence against the finding, such as a failed reproduction or a separate measurement. *Deferred or excluded* means that the finding was acknowledged but left unresolved for a stated reason, such as being out of scope, pre-existing, or intended behavior.

We call a task *test-blind* when the review reports a real defect even though the builder's test suite is green. Examples include a defect reproduced live, a test that passes without checking the relevant behavior, a broken coverage check, or tests that the builder never ran. As a simple extraction audit, a pattern search for phrases such as "found n issues" recovers 473 findings in 114 tasks, providing a lower bound on what the structured extraction should recover. A second coding by a third-vendor model is described in Section 4.3.1.

3.4 THREATS TO VALIDITY

Self-reported outcomes. The outcome labels come from the builder's own descriptions, and Karakaya et al. (2026) show that labels such as "fixed" and "wontFix" can reflect prioritization and workflow pressure as well as comment quality. Because we do not re-run reported repairs, *repaired* throughout the paper means *reported as repaired*, not independently verified as fixed.

No control group. The study covers one maintainer, one software environment, and one fixed model-role assignment, with no control group. A natural comparison would repeat the tasks with the two models' roles switched, but the recorded spend of $16,775 (mean $36.63 over the 458 tasks with a cost total) puts such a replay at roughly $17,800 per configuration.

Changing deployment. The protocol and infrastructure changed over time: in the last four weeks, 212 of 251 tasks were reviewed by delegation alone and 9 in conversation, returns were more often

truncated, and the framework gained new tools, so per-call behavior is more representative of the earlier period (Appendix B). The analysis plan was not fixed in advance.

Independent coding. The second coding of 60 tasks in Section 4.3.1 is a partial check rather than a replication: it identifies 222 findings against 279, and the count depends on how much detail each narrative gives, since grouped problems are not split into separate findings.

4 RESULTS

We examine four questions: whether the required review ran and how much of it is observable; what the reviewer did and what it cost; what the builder reported as findings, fixes, and rejections and how much those reports survive independent re-coding; and what later review rounds added. Two analysis populations are used: 431 of 486 tasks with event-level evidence of a reviewer switch or delegation, and 432 tasks whose final narratives are labeled completed and reviewed (416 belong to both). Unknown values are excluded from sums. Parent-side tool calls are attributed to the model requested for the call, and each positive increase in the cumulative cost counter to the model active when it was recorded. Detailed procedures and exclusions appear in Appendix C.

4.1 THE REVIEW IS VISIBLE IN THE LOG, BUT DELEGATED REVIEWS ARE ONLY PARTLY RECORDED

Event logs show reviewer activity in 431 of 486 tasks (88.7%), through the two mechanisms of Figure 1; this establishes that a review was started or requested, not that it completed, and 23 further tasks finished without starting one. The 431 reviewed tasks contain 635 candidate review passes, 99 by in-conversation switch and 536 by delegation, counting one `run_parallel` call as one pass (Appendix C).

The released database records the main agent's events in full but not the internal events of delegated reviewers, apart from six subagent records (Section 3.2); a delegated review appears as one tool call and one returned result. The September 8 backup preserves child records for 47 reviewer runs under 34 kept tasks. Because 347 of the 431 reviewed tasks use delegation and only 88 use an in-conversation switch, analyses of individual reviewer tool calls apply only to the latter group unless otherwise stated.

The reviewer almost never used write tools in the recorded inline reviews. The 88 in-conversation reviews contain 2,164 reviewer tool calls, of which three (0.14%) are write-tool calls, all writing review notes under a `tmp/` directory; none modified tracked source files. Excluding the 18 study-related tasks, or attributing calls from each task's recorded starting model instead, leaves the same three writes.

As a separate audit, a pattern search for potentially modifying shell commands produced 20 reviewer-window matches; manual inspection found that all 20 operated only on temporary directories or disposable Git repositories. The corresponding builder portion of the log contains 49,866 calls, of which 21.5% involve writes (21.7% after excluding calls made under another model). These figures establish behavior only for the recorded inline reviews. The 0.14% figure is therefore not an estimate of the write rate among delegated reviews.

The September 8 backup partially extends this analysis to delegation: its 47 delegated reviewer runs made 2,582 tool calls after switching to `gpt-5.6-sol`, 34 of them writes, all under `tmp/` (Appendix C.1). The three reviewer children retained in the released database behave the same way (one write, to `tmp/`). The remaining delegated reviews have no child-level record.

What the reviewer did. The recorded reviewer windows contain 1,429 shell commands. About half searched the code or related information, about one fifth inspected Git history or repository state, and 12.9% ran probe scripts or tests. These statistics describe the actions recorded in the log; they do not establish that any particular command found or confirmed a defect.

Many delegated review results are cut off. Of the 699 results returned by parallel subagents, 534 (76.4%) end at the fixed output limit of 3,021 characters. Among the 635 candidate review passes,

400 (63.0%) end with a truncated, timed-out, errored, or unrecorded return, or with a reviewer window still open at task end; none was counted as a completed review (Table 5).

4.2 THE COST RESULT RESTS ON 42 TASKS

We estimate reviewer spending in two complementary ways. First, per-step cost telemetry begins on August 15 and covers 299 tasks. For a task-level reviewer share we keep tasks with more than five usable cost records, a reconstructed total above $1, and no delegated review inside a requested reviewer window; eight tasks qualify. Reviewer spending accounts for 8.1% of their measured cost (median share 8.6%, median reviewer cost $1.37, range 6.4%–11.2%).

Attribution around model switches matters: 24 of the 798 cost records occur immediately after a switch and carry $21.23 of the $115.62 total. Assigning them entirely to the builder gives a reviewer share of 6.1%, and entirely to the reviewer 24.5% (Appendix C.1); we use the active-model attribution described above.

Second, the September 8 backup provides child-level costs for delegated reviews: each of the 47 reviewer children has a `task_history` row with the framework's cost total, and since a completed child's spend is added to its parent's total, the ratio gives the reviewer's share of that task (`scripts/reviewer_child_costs.py`). The children account for $177.84 against $1,898.22 for their parents (9.4%), with per-task median 14.6% (range 1.4% to 100%).

Combining the eight telemetry tasks with the backup-based child costs gives 42 tasks for which reviewer spending can be separated: $187.20 of $2,013.84, or 9.3%, covering 12% of the corpus's recorded spend. The eight telemetry tasks average $14.50 against the corpus mean of $36.63, so the subset is not representative of overall task cost, and all counters are framework telemetry rather than billing records.

The eight telemetry tasks all used the 50% reviewer limit, and each measured review share was below that limit. Among the 34 backup parents, 18 specified a 20% limit and 16 a 50% limit. The child cost exceeds the corresponding fraction of recorded parent spend in five tasks (three at 20% and two at 50%), but exceeds the recorded budget in none of the 10 parents with a recorded budget. Because narrated dollar amounts mix actual spending with budget limits, we do not use them. The cost and finding analyses also use different task sets, so we do not report a cost-per-finding measure.

4.3 THE TASK REPORTS CONTAIN MANY FINDINGS, BUT THE COUNT DEPENDS ON HOW THEY ARE READ

Among the 432 tasks with a completed review narrative, 347 (80.3%) report a positive number of findings, and the narratives describe 1,943 findings as confirmed, 1,822 of them outside the 18 study-related tasks (Table 1). Separately recorded outcomes include 1,779 reported fixes across 420 tasks with known fix counts, 19 rejected claims, 108 deferred or excluded items, and 148 tasks marked as having a verification gap (Table 7, Appendix C).

These outcome fields are not mutually exclusive and do not partition the 1,943 findings: in seven rows their component counts exceed the finding total, and fix counts are missing for twelve tasks. Among the 420 tasks with both counts known, the narratives report 1,779 fixes among 1,898 findings (93.7%; the ratio over all 1,943 would be 91.6%). We treat these as reported outcomes, not measures of repair correctness.

For context, Lin et al. (2026) report that developers rejected 56.3% of 31,073 CodeRabbit comments and accepted 36.4%. The comparison is descriptive only: the recipient here is another coding agent, and our 1.0% rejection rate is based on the builder's own narrative, whose labels carry the reliability limits shown by Karakaya et al. (2026). The 93.7% figure is the fraction of reported findings that the builder reported fixing, not an independently verified acceptance rate.

4.3.1 A SECOND READING OF 60 RANDOM TASKS

We randomly selected 60 reviewed tasks using seed 2027 and had a second model read them under the same extraction rules, plus a field for verification gaps, without access to the first tables. The second reader was a Google model (`gemini-2.5-pro` for seven of eight batches,

Table 2: Agreement between the first-pass extraction and an independent second coder on 60 randomly drawn reviewed tasks. "Within 1" counts exact matches and differences of one. Kappa is Cohen's κ on the binary fields. Sums are over tasks where the field is numeric for both coders.

Field	Comparable	Exact	Within 1	Sum (first / second)	κ
Review rounds	60	59	60	70 / 69	
Substantiated findings	59	45	52	279 / 222	
Reported fixes	59	48	54	223 / 198	
Rejected as false positive	58	57	58	2 / 1	
Deferred or excluded	58	54	56	10 / 6	
New findings in later rounds	43	42	43	5 / 4	
Review ran (binary)	60	60		60 / 60	
Verification gap (binary)	60	41		22 / 7	0.20
Clean final verdict (binary)	60	24		18 / 50	0.06

`gemini-3.1-pro-preview` for one), so the two readings come from different vendors, though both remain model coding rather than ground truth (Table 2; Appendix A).

Several fields are highly stable across the two readings: they agree on the number of review rounds in 59 of 60 tasks, on rejected findings in 57 of 58, on deferred or excluded findings in 54 of 58, and on whether a review ran in all 60. The finding count is less stable. The readers agree exactly on 45 of 59 tasks (76%) and within one on 52 (88%), but the second reader counts 222 findings where the first counts 279, lower in 13 of the 14 disagreements, mostly in a few narratives with many findings (one task is counted as 45 versus 17). The disagreements often turn on the unit of counting: eight stale docstrings could be eight findings or one, and the extraction rules do not resolve this case.

Extrapolating the second reader's counting style to the full dataset gives roughly 1,550 findings rather than 1,943, so we read the corpus as containing on the order of 1,500–2,000 builder-reported findings. The clean-verdict field is unreliable ($\kappa = 0.06$), conflating "nothing was found" with "everything found was fixed," and is not used; the verification-gap field ($\kappa = 0.20$) is analyzed separately below.

4.4 What kinds of problems did the reviewer report?

We sampled 200 task slots with probability proportional to each task's finding count (seed 2027); the second reader identified and classified 160 resolved findings using the functional/evolvability categories of Mäntylä & Lassenius (2009), as adapted by Beller et al. (2014), plus a *verification* category for tests and coverage checks (Table 8, Appendix C). Functional problems form the largest group, led by logic and then timing and concurrency; evolvability problems are mostly documentation that does not match the code. Among the functional/evolvability findings, 62% are functional, against roughly 25% reported for human reviewers. The comparison is descriptive only, since these are the builder's descriptions rather than adjudicated defects, and 40 of the 200 slots produced no complete enumerated finding.

4.5 Some findings exposed problems that the builder's tests did not catch

The first extraction marks 148 of 432 tasks (34.3%) as having a *verification gap*. The field combines the four situations defined in Section 3.3, so 34.3% is not a single defect rate. In the 60-task recoded sample the second reader marks 7 tasks versus 22 for the first, with five marked by both; of the second reader's seven, four involve defects reproduced after a green test suite, two vacuous tests, and one a broken coverage check. Because agreement is low ($\kappa = 0.20$), we treat these sample counts, rather than the corpus-wide 34.3%, as the stronger evidence.

The field needs clearer category boundaries before it can support a corpus-wide rate; Appendix A lists all 19 disagreements, and Appendix F examines three full-corpus cases with their diffs, including a coverage gate whose off-by-one excluded the fixed line from a report of 100% coverage. The vacuous tests and broken gate resemble the building-to-the-test behavior induced by Ma et al. (2026) under a hidden test oracle (Appendix D), though the settings differ and neither study establishes the pattern's frequency in production.

Checks based directly on the event log. Two analyses in Appendix C do not rely on narrative coding. Editing occurs after 75.1% of candidate review passes preceded only by passing test runs, but also after 78.9% of passes preceded by a failing run and 59.0% with no classified run, so these figures measure continued editing after a green suite, not exposed defects. Three reviewer-described mechanisms were also re-tested in bounded experiments: the reconstructed pre-edit code failed the corresponding check and the post-edit version passed, concrete examples that do not establish frequency.

4.6 Later review rounds find additional problems; how many were caused by earlier fixes is not measured

The first extraction identifies 106 tasks with more than one review round, all but eight with at most three. Ninety have a known number of later-round findings, and 54 of these (60.0%) report at least one new finding in round two or later, for 301 later-round findings in total (50.9% of all 106).

These counts are sensitive to how narratives are summarized; one narrative reports seven new round-two findings where the reviewer's own reports contain twelve. The 60-task agreement sample cannot validate the corpus total directly: only 43 rows have numeric later-round counts from both coders (sums 5 and 4, against about 30 expected at the corpus rate), and the remaining 17 tasks have a non-numeric value from at least one coder (`audit/coding/`).

We therefore re-coded the later-round field on a fresh random sample of 60 reviewed tasks (seed 2027), recording the count and the supporting sentence for each narrative (`scripts/later_round_recoding.py`, Appendix C.3). The sample contains 78 review rounds and 15 multi-round tasks, consistent with the corpus rate, and 24 later-round findings across 10 tasks, against about 41 expected; 4 of the 15 multi-round tasks report no new finding after round one. That yields 1.6 later-round findings per multi-round task, against 2.8 in the first extraction: below the original 301 total and far above the agreement sample's comparable rows. It remains a single model reading of builder narratives rather than ground truth.

No analysis here determines how many later-round findings were caused by earlier repairs. The second coding records a reported source for its 160 sampled findings: 83% from the task's own implementation work, 11% pre-existing, and 6% (95% CI 3–11%) from previous repairs, but only six of the 160 are later-round findings, so the 6% does not bound the repair-introduced share of the 301. The storage-engine case in Appendix F shows one repair introducing a defect that the next round found, consistent with prior observations of incorrect fixes (Yin et al., 2011; Smith et al., 2015; Le et al., 2019), without establishing their frequency here. More generally, the 301 later-round findings could be defects already present, reached as development continued, or introduced by repairs; the data do not distinguish these. Editing occurs after 71.7%, 64.0%, and 68.4% of first-, second-, and third-round candidate passes (Table 6), but progressively fewer selected tasks reach later passes, so these rates do not measure the incremental value of another round.

5 Conclusion and instrumentation for a controlled study

We analyzed post mortem the parent-side event log of a ten-week production deployment: one frontier model built software, and a second-vendor model reviewed it under a budget cap. Where review ran in conversation, the reviewer was almost entirely read-only (3 writes among 2,164 tool calls, none to tracked source), and on the 42 tasks where review spending could be isolated it accounted for 9.3% of task cost. The builder's narratives contain 1,943 reported findings, but a second coding found roughly one-fifth fewer; reported fixes were not re-executed, and the verification-gap label showed low agreement between coders.

These gaps motivate a small set of instrumentation requirements for a controlled study. *Link each finding to the call that produced it*: assign an identifier at report time and preserve it through repairs and later rounds. *Record child-agent work*: delegated reviewers account for most reviews here yet leave missing or truncated traces. *Record usage by role*, including subagents, so reviewer cost and budget compliance can be measured beyond the few tasks covered here. *Re-run repairs*: save the code before and after each repair with the failing reproduction, so reported fixes can be replayed and verified. Finally, *add a control arm*: replaying the tasks with the roles switched, at roughly $17,800 per configuration (Section 3.4), would show whether the workflow depends on the role assignment.

ETHICS STATEMENT

The trace database records the development activity of one identifiable maintainer, an author of this paper, who consented to its release for research. The records contain file paths, prompts, and repository content from that maintainer's machine; they contain no data about other people beyond names appearing in public open-source metadata. No human subjects were studied. One case study describes a code-execution vulnerability; the vulnerable code was repaired during the deployment, before this analysis, and the description follows the repaired public record. The paper measures a review protocol and does not release any capability beyond the trace data and analysis scripts.

STATEMENT ON THE USE OF AI

AI was involved in this work at four levels, and we separate them.

The subject matter is model-written: the studied code was produced by `claude-fable-5` and reviewed by `gpt-5.6-sol`, and the task narratives analyzed throughout are the builder model's own reports.

The first-pass finding extraction was performed by `claude-fable-5`, the same model and vendor as the builder whose narratives it read; the paper discloses this in Section 4.3 and treats the extracted counts as model-read self-reports. The second coding was performed by Google models (`gemini-2.5-pro` for seven batches, `gemini-3.1-pro-preview` for one), chosen so that neither the builder's nor the reviewer's vendor graded the builder's account. The complete later-round coding of the fresh 60-task sample (Section 4.6) was performed by `claude-fable-5-1`, again the builder's vendor; its per-task codes and quoted evidence are released (`scripts/later_round_codes.json`).

The analysis scripts released with the artifact were written with model assistance and were checked by running them against the database; their outputs are the numbers reported.

No completed independent human adjudication of findings, dispositions, or verification-gap labels is documented in the released artifact; the disagreement records in Appendix A are provided for such adjudication.

REFERENCES

Chris Adams, Arjun Singh Banga, Parveen Bansal, Souvik Bhattacharya, Payal Bhuptani, Rujin Cao, et al. Automating low-risk code review at Meta: RADAR, risk calibration, and review efficiency. *arXiv preprint arXiv:2605.30208*, 2026.

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eva Wang. Automated unit test improvement using large language models at Meta. *ACM International Conference on the Foundations of Software Engineering (FSE)*, 2024.

Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. *International Conference on Software Engineering (ICSE)*, 2013.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785. URL `https://philmcminn.com/publications/barr2015.pdf`.

Andrew M. Bean et al. Measuring what matters: Construct validity in large language model benchmarks. *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 AI on experienced open-source developer productivity. *arXiv preprint arXiv:2507.09089*, 2025.

Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? *Working Conference on Mining Software Repositories (MSR)*, 2014.

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, et al. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*, 2023.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Bo Chen. From runnable to verifiable: An independent reproducibility study of LLM/agent-driven vulnerability validation artifacts. *arXiv preprint arXiv:2608.09567*, 2026.

Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification: A concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

Kowshik Chowdhury, Dipayan Banik, K M Ferdous, and Shazibul Islam Shamim. From industry claims to empirical reality: An empirical study of code review agents in pull requests. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, 2026. arXiv:2604.03196.

Umut Cihan, Vahid Haratian, Arıda Arı, Buse Yıldız, Eray Tüzün, et al. Automated code review in practice. *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2025.

Shamse Tasnim Cynthia, Ratnadira Widyasari, Banani Roy, Ting Zhang, and David Lo. "go home Copilot, you're drunk": Understanding developer responses to agent-generated code review comments. *arXiv preprint arXiv:2607.21997*, 2026.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. TRAIL: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Liming Dong, Qinghua Lu, and Liming Zhu. AgentOps: Enabling observability of LLM agents. *arXiv preprint arXiv:2411.05285*, 2024a.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 2024b.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *International Conference on Machine Learning (ICML)*, 2024.

Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamerigts, Pierre-Antoine Manzagol, Stanislav Skoview, et al. Resolving code review comments with machine learning. *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2024.

Shashwat Goel, Joschka Struber, Ilze Amanda Auzina, Karuna K. Chandra, Ponnurangam Kumaraguru, Douwe Kiela, Ameya Prabhu, Matthias Bethge, and Jonas Geiping. Great models think alike and this undermines AI oversight. *arXiv preprint arXiv:2502.04313*, 2025.

Jianian Gong, Nachuan Duan, Ziheng Tao, Zhaohui Gong, Yuan Yuan, and Minlie Huang. How well do large language models serve as end-to-end secure code agents for Python? *arXiv preprint arXiv:2408.10495*, 2024.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. *International Conference on Learning Representations (ICLR)*, 2024.

Ryan Greenblatt, Buck Shlegeris, Kshitij Sachan, and Fabien Roger. AI control: Improving safety despite intentional subversion. *International Conference on Machine Learning (ICML)*, 2024.

Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. *International Conference on Software Engineering (ICSE)*, 2010.

Kunal Handa, Alex Tamkin, Miles McCain, Saffron Huang, Esin Durmus, Sarah Heck, Jared Mueller, Jerry Hong, Stuart Ritchie, Tim Belonax, et al. Which economic tasks are performed with AI? evidence from millions of Claude conversations. *arXiv preprint arXiv:2503.04761*, 2025.

Mike Helwig. Memory as infrastructure: Reliability engineering for persistent agent memory in months-long LLM-assisted development. *arXiv preprint arXiv:2609.05510*, 2026.

Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. *International Conference on Software Engineering (ICSE)*, 2013.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *International Conference on Learning Representations (ICLR)*, 2024.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *International Conference on Learning Representations (ICLR)*, 2024.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 435–445, 2014. doi: 10.1145/2568225.2568271. URL `https://www.cs.ubc.ca/~rtholmes/papers/icse_2014_inozemtseva.pdf`.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Anik Jha. Decorrelation is not complementarity: Skill, not lineage, governs trusted-monitor ensembles. *arXiv preprint arXiv:2608.16190*, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *International Conference on Learning Representations (ICLR)*, 2024.

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. *Working Conference on Mining Software Repositories (MSR)*, 2014.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S. Siegel, Boyi Wei, et al. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. *arXiv preprint arXiv:2510.11977*, 2025.

Veli Karakaya, Utku Boran Torun, Baykal Mehmet Uçar, and Eray Tüzün. Understanding the limits of automated evaluation for code review bots in practice. *arXiv preprint arXiv:2604.24525*, 2026.

Arsham Khosravani and Audris Mockus. Detecting AI coding agents in open source: A validated multi-method census of 180 million repositories. *arXiv preprint arXiv:2606.24429*, 2026.

Minseo Kim. tap: A file-based protocol for heterogeneous LLM agent collaboration. *Korea Computer Congress (KCC)*, 2026. arXiv:2606.14445.

Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. *International Conference on Software Engineering (ICSE)*, 2019.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. The rise of AI teammates in software engineering 3.0: How autonomous coding agents are reshaping software engineering. *arXiv preprint arXiv:2507.15003*, 2025.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-Bench illusion: When state-of-the-art LLMs remember instead of reason. *arXiv preprint arXiv:2506.12286*, 2025.

Hong Yi Lin, Mingzhao Liang, Patanamon Thongtanunam, and Kla Tantithamthavorn. Is agentic code review helpful? mining developers' feedback to CodeRabbit reviews in the wild. *arXiv preprint arXiv:2607.03316*, 2026.

Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. LLaMA-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2023.

Yanuo Ma, Ben Kereopa-Yorke, and Ben Schultz. Building to the test: Coding agents deliver what you check, not what you requested. *arXiv preprint arXiv:2606.28430*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM critics help catch LLM bugs. *arXiv preprint arXiv:2407.00215*, 2024.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? *International Conference on Learning Representations (ICLR)*, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *International Conference on Machine Learning (ICML)*, 2025a.

Ruwei Pan, Hongyu Zhang, and Chao Liu. CodeCoR: An LLM-based self-reflective multi-agent framework for code generation. *arXiv preprint arXiv:2501.07811*, 2025b.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Sheng-Wei Peng, Yi-Hsun Lin, and Yi-Pei Lee. Inference economics of enterprise coding agents: A case study of cloud vs. on-premise LLMs. *arXiv preprint arXiv:2607.13080*, 2026.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023.

Lucas Pinto. Calibration-family overfit: Why trusted sabotage monitors don't transfer across lineages. *arXiv preprint arXiv:2607.06596*, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. ChatDev: Communicative agents for software development. *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Jonan Richards, Kosei Horikawa, Youmei Fan, Yutaro Kashiwa, and Mairieli Wessel. AgentLogs: A dataset for opening the black box of GitHub's cloud agent. *arXiv preprint arXiv:2608.29204*, 2026.

Olawale Salaudeen, Anka Reuel, Sanmi Koyejo, et al. Measurement to meaning: A validity-centered framework for AI evaluation. *arXiv preprint arXiv:2505.10573*, 2025.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators. *arXiv preprint arXiv:2206.05802*, 2022.

Niruthiha Selvanayagam and Taher A. Ghaleb. AI-to-AI code reviews of GitHub pull requests. *arXiv preprint arXiv:2608.21311*, 2026.

Koushik Sen. KISS Sorcar: A stupidly-simple general-purpose and software engineering AI assistant. *arXiv preprint arXiv:2604.23822*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

Xinyuan Song, Zekun Cai, and Liang Zhao. When AI reviews its own code: Recursive self-training collapse in code LLMs. *arXiv preprint arXiv:2606.28438*, 2026.

Tao Sun et al. BitsAI-CR: Automated code review via LLM in practice. *arXiv preprint arXiv:2501.15134*, 2025.

Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota. Code review automation: Strengths and weaknesses of the state of the art. *IEEE Transactions on Software Engineering*, 2024.

Hanna Wallach, Meera Desai, Nicholas Pangakis, A. Feder Cooper, Angelina Wang, Solon Barocas, Alexandra Chouldechova, Chad Atalla, Su Lin Blodgett, Emily Corvi, et al. Position: Evaluating generative AI systems is a social science measurement challenge. *International Conference on Machine Learning (ICML)*, 2025.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. *arXiv preprint arXiv:2406.04692*, 2024.

Zuodong Xiang, Yike Zhang, YueMing Zhang, and Hailu Xu. Cross-model LLM code review: Should you use Claude to review Codex or vice versa? *Workshop on Agentic Software Engineering (KDD)*, 2026. arXiv:2607.21656.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

Yuxuan Zhu et al. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint arXiv:2507.02825*, 2025.

A SECOND-CODER DISAGREEMENTS ON THE VERIFICATION-GAP LABEL

Table 3 lists all 19 tasks where the two coders disagreed about the verification-gap label in the 60-task sample from Section 4.3.1. The full coding from both readers and the agreement results are included in the released artifact (`audit/coding/` and `audit/results/agreement_results.json`).

These disagreements help explain the result in Section 4.5. The second coder is not simply more strict. In several tasks that it did not label as a verification gap, its own notes still describe vacuous tests or tests that never exited. This suggests that the two coders often disagreed about how to use the label, not about what the task narrative said. We provide these records so they can be reviewed by a human later. They have not yet been manually checked.

Table 3: All 19 verification-gap disagreements on the 60-task sample. "First" and "second" are the two coders' labels; "kind" is the second coder's evidence category where it marked the task; notes are the second coder's, lightly compressed.

Task	First	Second	Kind	Second coder's note
`0b414cd9`	n	y	vacuous test	Reviewer found 3 test-hardening issues and 2 stale comments.
`0feac55d`	n	y	vacuous test	Review found stale test, discarded failure reason, and dropped toast.
`24d4b36d`	y	n	—	Moved tunnel-filter tests never called production code.
`2c0b7765`	y	n	—	Pairing allow-list bypass, state-filename collisions, approve-lock race.
`3f17722a`	y	n	—	
`4390c337`	y	n	unclear	
`4399fa3c`	y	n	—	
`6a32f76c`	y	n	—	Headless API client leaked a server-owned agent state per run.
`887de27d`	y	n	—	Initial flex-wrap made remote buttons overlap at widths of 360px or less in the running state.
`8a73b31b`	y	n	—	Popup exclusivity, Escape focus hand-off, duplicate editor-title controls.
`8b0a7da1`	y	n	—	Sweep firing after task end hid the finished task's panels.
`931e0719`	y	n	—	Live test hand-wrote panel DOM instead of exercising production renderers.
`a3f745b0`	y	n	—	New parameter inserted mid-signature, shifting historical positional arguments.
`b8cca5b8`	y	n	—	Object cache hashes CFLAGS, so PGO profiles were silently never consumed.
`c7f9dce7`	y	n	—	Non-scrollable summary element in the selector.
`cdde00e9`	y	n	—	Reap-inflight increment moved inside the reap mutex.
`d0791e25`	y	n	—	Dispatched cron sessions load the cron agent as a fresh synthetic module.
`d3fb4dbc`	y	n	—	The new JS test passed but never exited.
`e1b7f404`	y	n	—	Reviewer found 4 major issues, all fixed.

B COHORT DIFFERENCES ACROSS THE DEPLOYMENT

The deployment changed over time. Table 4 splits the 486 tasks into two-week periods by start time (Pacific time) and reports, for each period, how the reviews were run and how many delegated returns were cut off. `scripts/review_passes.py` computes the table.

The biggest change is how reviews were run. Early in the deployment, more reviews happened directly in the conversation. Later, most reviews were delegated to subagents, and the share of truncated parallel returns rose from 51% in the first two weeks to more than 80% in the last four. During the same period, the framework gained new tools, including a parallel command runner, a memory toolset, and background job control.

This means that the per-call reviewer results in the paper mostly describe the early deployment. The read-only compliance result, the action mix, and the eight-task telemetry cost analysis all come from the 88 inline-review tasks, and 79 of those started before August 19.

Table 4: Review mode and truncation by two-week period of task start, over all 486 tasks. The two review columns partition the 431 reviewed tasks of Section 3.1: the 88 tasks with an in-conversation review (including the four that also delegated) and the 343 tasks reviewed by delegation alone; the remaining 55 tasks had no review. Truncation shares are over `run_parallel` returns.

Period starting	Tasks	In conversation	Delegation only	Unreviewed	Parallel returns truncated
Jul 15	58	32	22	4	23/45 (51.1%)
Jul 29	78	27	40	11	55/79 (69.6%)
Aug 12	99	20	69	10	75/111 (67.6%)
Aug 26	99	7	71	21	114/132 (86.4%)
Sep 09	152	2	141	9	267/332 (80.4%)
All	486	88	343	55	534/699 (76.4%)

C ADDITIONAL ANALYSIS DETAIL

This appendix gives the procedures and detailed tables summarized in the main text. It covers the event-level audit, the classification of preserved review records, the event-based checks of the verification-gap label, and editing after each review pass.

C.1 EVENT-LEVEL AUDIT PROCEDURE

Corpus selection. Two task records in the released database (`236692e3`, `38c00763`) say how the selection of Section 3.2 was made; the selection script prints September 9, the UTC date of the 19:12 Pacific backup, whose last task started at 18:49. One released task (`4aeb9a23`) started at 18:40 that evening and is absent from the backup, so its row was written to the database after the copy was taken. All 63 dropped top-level tasks name both models in another phrasing, as do all 388 top-level tasks in the backup (4 of the 329 kept tasks are subagent records whose prompts repeat the parent's instruction block). Of the 63, 54 predate July 16 UTC, the released corpus's first day in the selection script's calendar, and 9 fall inside the window and were removed by the prompt string alone. With one kept child that also switched to the reviewer (`9ecff9cb`, August 8), the backup records 64 reviewer subagent runs before the trim. The released database holds six subagent records in all, three of them reviewer runs (`9ecff9cb` and the two September 8 database checks), so part of the opacity below the parent agent in Section 4.1 is a product of the selection. The screen is a prompt string and removed no task for what happened in it; the 55 unreviewed tasks of Section 3.1 are counted among the kept tasks only.

Window segmentation. For each task, we replay the event stream in sequence order and track the active model, updating it at each `set_model` call. Tool calls made while `gpt-5.6-sol` is active form reviewer windows; the remaining calls form builder windows. Subagent reviews cannot be divided this way because their events are not recorded separately. The reviewer-behavior audit therefore uses the 88 inline tasks.

Read-only compliance. The protocol requires the reviewer to leave the source tree unchanged. We count file-writing calls (`Write`, `Edit`, `NotebookEdit`) inside reviewer windows and inspect each one. We also scan reviewer `Bash` commands for operations that could change files, such as Git writes, in-place edits, deletions, and file moves. We inspect the matches by hand because some commands, such as copying a file into a temporary directory, match the pattern without changing the source tree.

Delegated reviewers in the backup. The September 8 backup (Section 3.2) keeps the events of the 85 subagent records that the trim removed, so the same write count can be made for the delegated reviewers recorded there (`scripts/backup_subagent_audit.py`). We take the first `set_model` call of each child as its role: 63 children switched to `gpt-5.6-sol` (two to `gpt-5.6-sol-xhigh`), 35 of them after one or two read or shell calls, 12 to the builder model, and 10 never switched. Of the ten, seven were dispatched for builder work and three carried a review prompt naming the reviewer model but made no `set_model` call: `e89a7861` (24 calls, no writes, under a third model), `118769be` (14 calls under the builder model, one write, a findings file in the working directory), and `c59841ab` (no tool call). One further reviewer child, `9ecff9cb`, carries the protocol string and was kept (42 calls after the switch, one write, under `tmp/`); it is counted in the released database, not among the 63. Of the 63 dropped reviewer children, 47 belong to 34 kept tasks; after the switch they made 2,582 tool calls, 1,716 of them shell commands and 34 of them write-tool calls, every one to a path under a `tmp/` directory. The other 16 reviewer children, dated July 9 to September 1, belong to tasks outside the corpus (eight dropped top-level tasks and five parents absent from the backup). They made 123 write-tool calls, 99 of them to tracked source files, all in four runs on July 14, the day before the first released task, whose prompts asked for "review and debug" and carried no read-only rule. The shell-command scan of the previous paragraph was not repeated for these children.

Action classification. We classify reviewer `Bash` commands as search (grep and related commands), Git inspection (diff, show, log, status), file reading, probe scripts (inline Python), test execution, artifact checks, and network fetches. The released scripts contain the keyword rules used for these categories. The shares of probe and test commands show how much of the review runs code versus reads code.

Cost attribution. For tasks with usage telemetry, we assign each positive increase in the cumulative cost counter to the model active when that sample was recorded. The counter sometimes decreases; the released script counts and reports these decreases rather than treating them as spending, and none occurs in the clean subset below. Subagent dispatches make this attribution less reliable because a subagent's cost appears inside whatever window is active when the subagent runs. We therefore report reviewer cost shares only for inline tasks whose reviewer windows contain no `run_parallel` or `run_agent` call. This clean subset contains 8 tasks. Twenty-four of their 798 cost records occur immediately after a model switch and together account for $21.23 of the $115.62 total. Assigning all of it to the builder gives a reviewer share of 6.1% ($7.06); assigning all of it to the reviewer gives 24.5% ($28.29); the event-by-event assignment, which follows the order in which the models ran, gives 8.1% ($9.36). For one of the eight tasks the telemetry total is 13% above the cost stored in the task record and is used as the denominator; the other seven match exactly. For the delegating tasks, the reviewer's cost is the child's `task_history` cost total in the backup and the denominator is the parent's total in the released database, which the framework increments with each finished child's spend (`scripts/reviewer_child_costs.py`); the parent `1e7f3b9c` was still running when the backup was taken, so its backup row carries no cost. None of the 34 parents also switched to the reviewer in conversation, and none is among the eight telemetry tasks. The largest share, 100%, is a $2.94 task (`eecf4ea4`) whose recorded total equals its child's cost, so the parent's own spend is not in its counter. The task narratives also report dollar amounts near the word "review" (94 tasks), but Section 4 shows that these amounts mix actual spending with budget allowances, so we do not use them as additional evidence.

Preserved delegation returns. Across all delegation returns, including non-review work, 534 of 699 parallel returns (76.4%) and 48 of 115 single-agent returns (41.7%) contain a truncation marker. Every marked return is exactly 3,021 characters long, and no return is longer. This matches a fixed truncation to the first and last 1,500 characters around a 21-character marker. Thus, the 76.4%

Table 5: Preserved record for the 635 candidate review passes in the 431 reviewed tasks. "Unadjudicated" means this study has not determined whether the record constitutes a completed assessment. One delegation call counts once, however many reviewer children it dispatched.

Preserved record for a candidate pass	Count
Delegated return recorded, truncated at 3,021 characters	336
Delegated return recorded, unadjudicated	139
Closed in-conversation interval, unadjudicated	96
Error return	42
Timeout return	14
No return recorded	5
In-conversation interval still open at task end	3
Total	635

rate reflects a fixed output limit rather than occasional overflow. In the framework's logging code (release 2026.9.8, inspected after the deployment window), truncation is applied when the tool result is recorded, while the full return is passed to the parent model. The truncation therefore limits what was saved, not what the builder saw. We did not inspect the code deployed during the window.

Candidate review passes. A candidate review pass is one `set_model` call that switches the conversation to the reviewer model, or one `run_parallel` or `run_agent` call whose arguments name `gpt-5.6-sol`. One call is one pass, so a `run_parallel` call with several reviewer children counts once. The 431 reviewed tasks contain 635 candidate passes: 99 in-conversation switches and 536 delegations. An in-conversation pass is closed when a later `set_model` leaves the reviewer model and open when the task ends inside the reviewer window. A delegated pass is classified by the tool result returned to the parent: truncated at 3,021 characters, timed out, errored (the result is flagged as an error, starts with an error message, or reports an unhandled exception or a failed child), or recorded in full. `scripts/review_passes.py` computes every number in this subsection.

Table 5 classifies the 635 candidate passes by what was preserved in the record. Of these, 400 (63.0%) end with a truncated, timed-out, errored, or unrecorded return, or with an open reviewer window. None has been judged in this study to be a completed assessment. A timeout does not necessarily mean that the assessment was lost: in one case, the parent later retrieved the assessment from a timed-out delegation.

Editing after passing tests. This event-based analysis does not depend on task narratives. A test run is a builder `Bash` call that invokes a test runner (`pytest`, `npm test`, `npm run test`, `vitest`, `jest`, `mocha`, `node --test`, `cargo test`, `go test`, `ctest`, or `unittest`). Its outcome is failing when the result is flagged as an error or reports a failure count, passing when it reports a pass count and no failure, and unclassified otherwise. Each pass is grouped by the classified runs between the previous pass (or the task start) and the pass itself. Of 177 candidate passes preceded only by passing runs, 133 (75.1%) are followed by at least one file-writing call (`Write`, `Edit`, `MultiEdit`, or `NotebookEdit`) by the builder before the next pass or task end. The corresponding rates are 78.9% (142 of 180) when at least one preceding run failed and 59.0% (164 of 278) when no classified run exists. Across all three groups, 439 of 635 passes (69.1%) are followed by an edit. The rate after passing-only runs is within a few points of this base rate, so continued editing is the ordinary state of these tasks rather than a signal specific to green suites.

There are two limits to this measure. The run classifier uses reported pass and fail counts only; it does not link a test run to the revision that was reviewed. And the intervals begin when review is invoked, before any feedback has arrived. The edits are not linked to individual findings. The result therefore measures continued editing after recorded checks, not whether the review was effective.

Reproduced mechanisms. Bounded reproductions provide the strongest evidence for specific defects. In one report-viewing task, a recorded coverage check reports that all 63 feature lines were covered. The reviewer switch occurs about a hundred events later, followed by repair edits to the same file. The reviewer identifies two runtime problems: failure-result strings are treated as successful report writes, and parent-directory paths are classified as belonging to the report folder. Four adverse probes fail their correctness checks on the reconstructed pre-edit functions and pass after the

Table 6: Event-derived candidate passes followed by editing activity in all 431 reviewed tasks. Each row counts the tasks that reach that pass ordinal. Pass ordinals differ from narrative round counts and do not certify completed assessments or correct repairs.

Pass ordinal	Tasks reaching pass	With edits	Share
First	431	309	71.7%
Second	136	87	64.0%
Third	38	26	68.4%
Fourth	11	6	54.5%
Fifth or later	19	11	57.9%

Table 7: Narrative-derived counts from the first-pass extraction. Unknown values are omitted from sums. Disposition counts are separate extractions, not mutually exclusive parts of one verified inventory, and their denominators differ.

Narrative-derived quantity	Count
Completed tasks labeled as reviewed	432
Tasks with a positive numeric finding count	347
Findings summed over 421 known task totals	1,943
Reported fixes summed over 420 known task totals	1,779
Reported rejected claims	19
Reported deferred or excluded items	108
Tasks labeled with a verification gap	148

logged edits. A successful-write control passes in both versions. The rendering boundary is stubbed, and the path result shows incorrect classification rather than a demonstrated filesystem escape.

A third reproduction concerns an updater that marks a rebuilt tracked artifact with Git's `skip-worktree` flag. The modification disappears from ordinary status output, but resetting to a later release that changes the artifact fails with exit code 128. Clearing the flag allows the reset to succeed.

These reproductions show that selected reviewer-reported defects survived the checks used in the original workflow. They do not give a population rate. Reviewers finding problems that tests miss is a stated expectation of code review in Bacchelli & Bird (2013); the contribution here is the trace-level demonstration in a model-reviews-model setting, not the general claim that code review has this property.

Continued editing by pass. The event analysis shows editing after 71.7% of first candidate passes, 64.0% of second passes, and 68.4% of third passes (Table 6); 30 passes are fourth or later. Fewer tasks reach later passes, and those tasks are a selected group. Reaching another pass also depends on what happened earlier, rather than on a randomized assignment. These rates therefore do not measure the additional benefit of another review.

C.2 Six-case analysis

A complementary six-case analysis codes 22 items, including 20 related to the reviewer. The other two are an issue found by the implementer before review and a finding from an external auditor. The analysis separates possible runtime defects, testing gaps, and non-defect items, and links repeated reports to their finding histories.

Three selected mechanisms are tested through bounded reproductions: two JavaScript functions reconstructed from logged code and edits, and a Git update failure reconstructed in disposable repositories. These experiments test specific failure conditions rather than complete historical application repairs. The case coding was done by an analyst and is provisional; it has not received independent human review.

Table 8: Class of 160 randomly sampled narrated findings, coded by the second coder, against the human code-review baseline. Human figures are the share of review findings concerning evolvability reported by Mäntylä & Lassenius (2009) (two industrial and student datasets) and Beller et al. (2014) (two open-source projects); neither scheme has a verification class, so the human column is given on the functional/evolvability axis only. 95% intervals are Wilson.

Class	This corpus ($n = 160$)	Share (95% CI)	Human review
Functional	85	53% (45–61)	about 25%
Evolvability	52	33% (26–40)	about 75%
Verification (tests, gates)	17	11% (7–16)	not coded
Non-defect	6	4% (2–8)	
Functional share of functional+evolvability	85/137	62% (54–70)	about 25%

C.3 Finding taxonomy: full account

To describe what the reviewer reported at the finding level, we drew 200 finding slots with probability proportional to each task's first-pass finding total (seed 2027; 111 distinct tasks), so each narrated finding had the same chance of being selected. The second coder then listed every finding in each selected task in the order it was narrated and classified it.

The classification uses the functional/evolvability split from Mäntylä & Lassenius (2009), which Beller et al. (2014) applied to open-source review comments. It uses the functional subclasses from that scheme (logic, check, interface, resource, timing, security, support, requirement) and the evolvability subclasses (documentation, structure, solution approach). Because this builder-reviewer loop also reviews tests and gates, we add a third class, *verification*, for findings about test or gate machinery. We also add a *non-defect* class for items the narrative records as rejected, intended behavior, or out of scope. The k-th finding listed for each selected task is the sampled unit.

Of the 200 draws, 160 resolved to an enumerated finding. The other 40 were unresolved for two reasons: 32 went beyond the second coder's finding count in tasks where it listed at least one finding, and 8 came from tasks that the second coder coded with zero findings. Three enumerations were also incomplete relative to the narrated counts (12 versus 5, 23 versus 5, and 8 versus 3). Thus, part of the gap comes from unfinished enumeration rather than only from a difference in the sampling unit.

The 40 unresolved draws come from 21 tasks, 7 of them multi-round. The storage-engine task in Appendix F contributes two. Their first-pass notes include both list-style documentation items ("8 stale docstrings," "26 findings … in INVARIANTS.md") and multi-round functional defects, so the sample misses some of both types. Only 6 of the 160 resolved draws are later-round findings. Draws were proportional to the first-pass finding total, and later-round findings are 301 of the 1,943 in that table (15.5%), so 160 draws would hold about 25 of them if they resolved at the same rate as the rest. Later-round findings among the 40 unresolved draws (7 of the 21 tasks with unresolved draws are multi-round) would account for part of the difference; the rest is unexplained. Section 4.6 gives the matching computation for the 60-task sample of Table 2, whose 43 comparable rows sum to 5 and 4 later-round findings.

Complete later-round coding of a fresh sample. Because 17 rows of that sample carry no number from one coder, we coded the later-round field once more on a fresh sample: `random.Random(2027).sample` over the sorted ids of the 431 tasks with an event-derived candidate review pass (Section 4.1). One reader (`claude-fable-5-1`) read each task's final narrative and recorded the number of narrated review rounds, the number of new problems the reviewer first reported in round two or later, and the sentence that supports both, verbatim (`scripts/later_round_codes.json`; `scripts/later_round_recoding.py` redraws the sample, checks every quoted sentence against the database, and prints the totals). The rules follow Section 3.3 and add the counting of an enumerated list: a single-round task, or a later round that only confirmed the fixes, scores zero; a narrated number is used as given; an enumerated list is counted; refinements or tests that a later round asked for count as problems; a later round said to have found new problems without a number or a list is unknown. The 60 tasks have 78 rounds; 15 tasks have more than one (the corpus rate of 106 in 432 predicts 15). Of the 15, 4 report no new problem after round one, one is unknown (`cf055d84`: round two reports "7 resolved, 4 partly"

and a list of wording issues without saying which are new), and 10 report 24 later-round findings in all (largest 5, in a paper-editing task whose focused re-review found five residual precision issues). The 59 numeric rows predict about 41 at the corpus rate; per multi-round task the sample has 1.6 against the first extraction's 2.8. The two samples may overlap; the ids of the first sample are in `audit/coding/`.

Table 8 gives the result. Functional findings are the largest group: 85 of 160 (53%). Logic is the largest subclass (47), followed by timing and concurrency (15), check and validation (7), resource and lifecycle (7), interface and wiring (5), and security (4). Evolvability findings account for 52 (33%), including 43 documentation and claim-drift findings (comments, docstrings, READMEs, or result narratives that do not match the code) and 9 structure or style findings. Verification findings account for 17 (11%): 5 vacuous tests, 5 missing tests, 5 broken gates, 1 flaky test, and 1 build check.

Among only the functional and evolvability findings, 62% are functional (95% CI 54–70%), compared with roughly 25% reported for human reviewers by Mäntylä & Lassenius (2009) and Beller et al. (2014). The sampling issue above does not remove the difference: even if all 40 unresolved draws were evolvability findings, the functional share would be 85 of 177 (48%), still about twice the human figure. Across all 707 findings the second coder listed in the 151 coded tasks (a convenience population, not a random sample), the split is 56% functional, 29% evolvability, 12% verification, and 3% non-defect.

There are two important cautions about the human comparison. The human studies coded reviewer comments, while we coded the builder's description of the reviewer's report. A builder may describe defects it fixed in more detail than style issues it ignored. Also, the deployment prompt asked for "review and debugging," so the reviewer was explicitly asked to look for defects. With these limits, the result is that the prompted cross-vendor reviewer in this deployment reported mostly functional defects, followed by documentation drift, with test and gate problems forming a separate group.

C.4 Analysis populations and measurement detail

This section gives the populations and measurement rules behind the questions listed at the start of Section 4. We answer them using retrospective trace analysis and bounded diagnostic reproductions. The deployment has no same-model or no-review control arm, so these analyses describe the observed workflow rather than measure the causal benefit of model diversity.

Analysis populations. The corpus contains 486 tasks and 1,078,323 events (July 15 to September 21, 2026, Pacific time). We separate tasks with event-level evidence of a reviewer switch or delegation from tasks whose result narratives are labeled completed and reviewed. The event-based population contains 431 tasks and the narrative-based population 432. Their intersection contains 416 tasks: invocation and narrated review use different inclusion rules, so 15 tasks appear only in the event-based group and 16 only in the narrative-based group.

We use the event-based population for reviewer participation and the narrative-based population for finding outcomes. We do not move counts from one population to the other. The narrative extraction covers all 486 task records. Unknown values are left out of numerical sums rather than treated as zero. The second coding and the finding taxonomy are sampled studies (60 tasks and 200 finding slots); the bounded reproductions concern three individual tasks.

Protocol and activity measurements. The artifact divides parent-side tool calls using the requested model switches. Within reviewer windows, it counts direct write-tool calls, scans shell commands for possible mutations for manual inspection, and classifies commands using keyword rules as search, Git inspection, file reading, probes, tests, and related actions. The complementary window includes every remaining tool call rather than a verified builder trace: 779 calls (1.6%) occurred in tasks that started on, or switched to, a model other than the builder. Delegated child execution is not separately recorded in the released database apart from six retained child task records; the September 8 backup adds 85 more (Section C.1). We also inspect delegation-return truncation and the evidence available for each pass to measure how complete the preserved record is.

Cost measurement. For tasks with usage telemetry and no delegation inside a reviewer window, the artifact assigns each positive increase in cumulative cost to the model active when the sample was recorded. Samples are emitted after each model response and before that response's tool calls execute. Thus, the increment recorded after a switch request is assigned to the first response from the newly active model. This endpoint rule follows the ordering in the log.

The replayed telemetry total is not always identical to the task's recorded cost. We exclude the separate narrative-dollar extraction because it mixes budget allowances and caps with actual expenditure.

Findings and verification. The fixed-schema narrative extraction records findings described as substantiated, reported fixes, rejected claims, deferred or excluded items, verification gaps, and later-round findings. These are model-extracted accounts of what the builder reported, not independently checked outcomes.

In particular, the field named `test_blind` accepts several kinds of evidence, including defects found after passing tests, vacuous tests, faulty coverage gates, and checks that the builder did not run. We therefore treat it as a broad verification-gap label. It does not always mean that a defect was independently demonstrated on the exact revision that passed the tests.

A complementary six-case analysis and three bounded reproductions are described in Appendix C. The case coding was done by an analyst and remains provisional because it has not received independent human review.

Repeated review. We report later-round outcomes for tasks with at least two narrated rounds and a determinate later-round finding count. Separately, the event-based analysis orders candidate passes (acknowledged reviewer intervals or reviewer-naming delegations; parallel children count once) and measures editing afterward. An invocation does not necessarily produce a completed assessment. Both analyses are descriptive because reaching another round depends on what happened in the preceding workflow rather than on a randomized assignment.

D Extended related work

This appendix preserves the full related-work discussion; Section 2 in the main text condenses it.

D.1 Multi-agent coding frameworks and self-correction

Role-based frameworks assign builder, tester, and reviewer roles to language-model agents and measure the gain on code-generation benchmarks. MetaGPT encodes standard operating procedures across roles (Hong et al., 2024), ChatDev organizes a simulated software company (Qian et al., 2024), and CAMEL studies role-play between communicating agents (Li et al., 2023). Later systems refine the pattern with dedicated test agents and repair loops, including Self-Collaboration (Dong et al., 2024b), AgentCoder (Huang et al., 2023), MapCoder (Islam et al., 2024), and CodeCoR (Pan et al., 2025b). A parallel line asks whether a model can critique itself. Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and CRITIC (Gou et al., 2024) report gains from iterated self-feedback, while Huang et al. (2024) show that self-correction without external signal is weak. Olausson et al. (2024) find that feedback from a stronger model outperforms self-repair, and trained critics catch bugs that both humans and the authoring model miss (Saunders et al., 2022; McAleese et al., 2024). Debate and ensembling offer further variants (Du et al., 2024; Wang et al., 2024; Burns et al., 2023).

These systems are evaluated on benchmarks, almost always with a single underlying model playing every role, and the unit of measurement is pass rate. We study a deployed two-vendor instance of the same architecture over ten weeks of production work. Our unit of measurement is the individual reviewer finding, with its disposition, cost, and round structure.

D.2 Cross-model oversight and AI control

The AI-control literature formalizes our setting: an untrusted model produces code and a second model monitors it under an audit budget (Greenblatt et al., 2024). Follow-up work disagrees on

whether vendor diversity helps the monitor. Pinto (2026) report that off-lineage monitors catch fewer planted backdoors at matched audit cost, while Jha (2026) find that monitor skill, and not lineage, drives detection. Related evidence motivates a second vendor: models favor their own generations (Panickssery et al., 2024), capability-correlated models make correlated errors that undermine oversight (Goel et al., 2025), models repair other models' vulnerable code better than their own (Gong et al., 2024), and pure self-review degenerates without an exogenous check (Song et al., 2026). Closest to our setting, Xiang et al. (2026) run a controlled cross-vendor review experiment on 116 LiveCodeBench problems with Claude Opus 4.7 and Codex GPT-5.5 in six conditions (solo, self-review, and both cross-model orderings). The reviewer sees the problem and the draft, cannot run code, and its revised program is the final submission. Claude reviewing Codex raises the pass rate from 71.6% to 89.7%; Codex reviewing Claude lowers it from 91.4% to 82.8%. By vendor, the direction that hurt is the direction our deployment ran, with an OpenAI reviewer over an Anthropic builder. Section 2 lists the differences between the settings (a reviewer that can execute probes, findings adjudicated by the builder rather than submitted directly, maintenance tasks without a hidden oracle) and notes that our record, having no comparison arm, can neither confirm nor overturn their result. Kim (2026) reports the one earlier deployment of a heterogeneous builder-reviewer pair: a 27-day, 37-generation self-applied operation in which Claude and Codex agents developed and reviewed the tap protocol, yielding 209 pull requests and 717 operational artifacts. Of 375 review artifacts, 69.8% of those from heterogeneous pairs recorded at least one defect or requested change, against 53.1% from homogeneous pairs. The study therefore has the same-vendor arm we lack; it does not report whether the flagged defects were real, what the reviewer did to find them, or what review cost.

All of this evidence is either synthetic red-teaming or a controlled benchmark, with the exception of Kim (2026), which reports outcomes without event-level instrumentation. Our traces come from benign production use, and they are complete enough to verify the protocol itself: we measure write-action leakage in reviewer turns, the mix of static and dynamic verification actions, and adherence to the budget cap. Our deployment has no same-model control arm, so it cannot say whether vendor diversity itself is what helps. We list the logging a future deployment would need to make that comparison.

D.3 LLM CODE REVIEW IN PRODUCTION

Automated code review predates agents. CodeReviewer (Li et al., 2022) and LLaMA-Reviewer (Lu et al., 2023) generate review comments from pretrained models, and Tufano et al. (2024) document the quality limits of this generation of tools. Industrial deployments now operate at scale: Google resolves reviewer comments with learned edits (Frömmgen et al., 2024), Meta gates low-risk diffs through a review funnel (Adams et al., 2026) and hardens tests with LLM assistance (Alshahwan et al., 2024), and ByteDance and industrial partners report production reviewer bots with adoption metrics (Sun et al., 2025; Cihan et al., 2025). Observational studies mine the resulting artifacts. Lin et al. (2026) classify developer reactions to 31,073 CodeRabbit review comments on 10,191 pull requests and find that 56.3% were rejected against 36.4% accepted (Section 4.3 contrasts this with the builder's 93.7% narrated fix rate); Cynthia et al. (2026) study why agent comments go unresolved; Selvanayagam & Ghaleb (2026) document a growing closed loop in which AI reviews AI-authored pull requests. Chowdhury et al. (2026) compare 3,109 AIDev pull requests in the commented review state and find that pull requests reviewed only by code-review agents merge at 45.2%, against 68.4% for human-only review, with most closed agent-only pull requests carrying low-signal comments; their outcome is a merge decision by a human, where ours is the builder's own account of what it fixed. Karakaya et al. (2026) show that developer-action labels such as fixed or wontFix track workflow pressure and are unreliable as ground truth for comment quality.

Two things separate our study from this line. First, in these deployments the reviewed code is written by humans, or the review is consumed by humans; in ours, one model writes everything and another model reviews it, so the finding-to-repair loop closes without a person in between. Second, the observational studies see comments and merge metadata, while we see the internals of both agents, including each review action and its cost. The concern raised by Karakaya et al. (2026) applies to our disposition labels too, and our method section treats them as reported outcomes.

D.4 Traces and case studies of deployed coding agents

Several corpora release agent trajectories, including SWE-Gym (Pan et al., 2025a), SWE-smith (Yang et al., 2025), and traces from simulated organizations (Xu et al., 2024); all are generated on benchmark tasks. At the other end, AIDev collects hundreds of thousands of agent-authored pull requests in the wild (Li et al., 2025; Khosravani & Mockus, 2026), and the Anthropic Economic Index characterizes usage in aggregate (Handa et al., 2025). Between these, AgentLogs (Richards et al., 2026) releases the process behind the pull requests: 64,255,174 session-log entries from 549,239 sessions of GitHub's Copilot cloud agent in 35,810 public repositories, each entry recording prompts, intermediate reasoning, tool calls, and token usage, for an agent that both opens and reviews pull requests. Longitudinal single-deployment case studies are emerging: Helwig (2026) instrument a months-long production coding-agent project around its memory subsystem, and Peng et al. (2026) analyze one developer's cost telemetry over two 28-day windows. Randomized studies measure productivity effects of coding assistants (Peng et al., 2023; Becker et al., 2025) but release no traces.

Our corpus sits between these. The pull-request corpora have breadth and no internals. The benchmark corpora have internals and no deployment. The single-deployment case studies have both for one agent and contain no review workflow. AgentLogs has breadth, internals, and deployment, and its agent does review pull requests, so it is not true that no released event-level record contains review. What it lacks is our protocol: its agent is a single product, review is one of several activities it performs on request, and no fixed second-vendor reviewer is paired with a builder on the same task. Our claim is correspondingly narrower. Ours is, to our knowledge, the first released event-level record of a cross-vendor builder-reviewer protocol, in which one vendor's model builds and another's reviews the same change, operating on production work with per-step costs attributed to each role. AgentLogs is also the natural place to test whether the action-mix checks of Section 4.1 transfer to a platform-scale agent. The deployment runs on the KISS Sorcar framework (Sen, 2026), whose developer is an author of this paper and the maintainer whose database we analyze (Section 1). The framework paper describes the review protocol but reports no measurement of it; the measurements in Section 4 are the first for this deployment.

D.5 Evaluation validity and defect dispositions

A growing methodological literature asks what evidence licenses what claim. Validity-centered frameworks (Salaudeen et al., 2025; Wallach et al., 2025; Bean et al., 2025), audits of agentic benchmarks (Zhu et al., 2025; Kapoor et al., 2024; 2025), and re-execution studies of published artifacts (Chen, 2026) all quantify gaps between reported and supportable results. For coding specifically, SWE-bench underestimates this gap: solution leakage and weak tests inflate resolved rates (Jimenez et al., 2024; Aleithan et al., 2024; Liang et al., 2025), echoing the older patch-overfitting result that plausible patches are frequently wrong (Le et al., 2019). Ma et al. (2026) show the same gap for agents rather than benchmarks: two production coding agents re-implementing a component under a hidden 222-test oracle reached near-perfect scores while leaving the requested library dead or absent, which the authors call building to the test; Section 4.5 relates this to the vacuous tests and broken gate in our sample, where the builder wrote the checks it then satisfied. Trace-derived failure taxonomies exist for multi-agent systems (Cemri et al., 2025; Deshpande et al., 2025), observability frameworks prescribe what agents should log (Dong et al., 2024a), and software engineering has classified defect dispositions since orthogonal defect classification (Chillarege et al., 1992), with later work showing that which bugs get fixed, and how they get labeled, reflects process as much as substance (Guo et al., 2010; Herzig et al., 2013; Kalliamvakou et al., 2014).

We inherit the disposition idea from this lineage and extend it where trace data forces an extension: alongside repaired, deferred, and rejected findings, our schema records what the log cannot show, such as repairs never independently re-executed and findings not linked to the review call that produced them. The audits of Zhu et al. (2025) and Chen (2026) target benchmarks and published artifacts. We apply the same discipline to a live deployment, and we turn the gaps that survive into logging requirements for controlled follow-up studies.

E BACKGROUND

E.1 CODING AGENTS AND THE EXECUTION FRAMEWORK

A coding agent combines a language model with tools and an execution loop. The model selects actions such as inspecting a file, editing code, or running a command; the framework executes those actions and returns observations that inform subsequent actions. ReAct describes this interleaving of reasoning and interaction with an external environment Yao et al. (2023). In software development, the resulting output is an evolving repository state together with the checks and explanations produced during the task, rather than a single generated program.

KISS Sorcar is the execution framework used in this study. Its architecture combines budget-tracked agent execution, continuation through summarized sub-sessions, coding and browser tools, parallel sub-agents, persistent conversation history, and Git worktree isolation Sen (2026). The framework is distinct from the models it invokes: it manages tool execution and task state, while a selected model supplies the next response or action. This distinction matters when interpreting traces because a task can span multiple model invocations, context continuations, and delegated executions.

E.2 THE BUILDER–REVIEWER WORKFLOW

We use *builder* and *reviewer* to denote roles in the workflow. The builder implements the requested change and responds to feedback. The reviewer examines the work, identifies possible defects or unmet requirements, and may use tests or diagnostic probes to investigate them. In the protocol studied here, the builder is instructed to use `claude-fable-5`, while review is assigned to `gpt-5.6-sol` under a budget cap. These are the model identifiers recorded by the deployment. The reviewer's read-only instruction limits its authority to change the implementation; it does not restrict the reviewer to reading source code. Whether execution follows that instruction is an empirical question.

The conceptual workflow is implementation, review, response, and possible re-review. A response may repair a reported problem, dispute its premise, or defer it with an explanation. Further review can examine the changed code and remaining concerns. Heterogeneous review assigns different model identities to the two roles, but that assignment alone does not establish independent errors or better detection.

Model identity and reviewer context are separate properties. Switching models within a continuing conversation gives the reviewer access to retained conversation context, including the builder's account of its work. Delegating review creates a separate execution whose information depends on the supplied instructions, accessible artifacts, and context passed to it. A model switch is therefore not equivalent to a fresh assessment without the builder's prior explanations. We distinguish these mechanisms when reconstructing review activity.

E.3 TESTS, FINDINGS, AND REPAIR EVIDENCE

Testing requires both exercising a program and determining whether its behavior is correct. The latter is the test-oracle problem: an oracle supplies the criterion against which an observed result is judged Barr et al. (2015). Passing tests establish that the exercised behavior satisfies those checks under the tested conditions. Line coverage measures execution of code, not whether assertions express the intended behavior. Empirical work on large Java systems found only low-to-moderate correlation between coverage and mutation-based test effectiveness after controlling for suite size Inozemtseva & Holmes (2014). Thus, coverage and correctness are distinct properties.

This distinction matters when a builder produces both a change and its verification. A mistaken interpretation of a requirement can appear in both the implementation and the expected test results. A reviewer can examine those assumptions, the adequacy of the tests, and the implementation itself. This motivates review without assuming that the reviewer supplies a correct oracle.

A reviewer finding is initially a claim about a possible problem under some condition. Its *disposition* records how the builder responds; its *validity* concerns whether the alleged problem exists. Accepting a report or making an edit does not by itself establish that the finding was valid or that the repair was correct. Stronger evidence links the reported condition to the relevant code and demonstrates

that the failure occurs before a change and is removed afterward. Repeated reports may concern the same unresolved condition, while a repair may introduce a different failure. These distinctions motivate tracking reports, responses, and validation evidence separately in the analyses that follow.

F CASE STUDIES

This appendix presents three cases referred to in Section 4.5: a coverage gate with an off-by-one error that excluded the line containing the fix, a repair that needed two review rounds to become correct, and a code-execution vulnerability found during a routine review. All quotes and code come from the task records in the released database. We identify each task by the first eight characters of its ID.

F.1 THE COVERAGE GATE THAT EXCLUDED THE PATCH LINE

Task `12bbab41` asked the builder to make a chat view resume auto-scrolling when a new task starts. The builder found the root cause: a stale scroll lock was not cleared when the run state changed. It added the one-line fix shown in Figure 2. It then wrote eight end-to-end tests against the production page, checked that six fail without the fix, and added a V8 coverage gate that reported 100% line coverage for the changed regions.

The defect was in the code that measured coverage. The reviewer inspected the gate and found the off-by-one error shown in Figure 2: `n < b - 1` excluded the final line of every marked region. In this case, that was the line containing the fix. The coverage check therefore did not require that line to run. This is a clear example of *test-blindness*: the builder's own check for test coverage contained a bug. If left in place, it could give incorrect coverage results for later changes that rely on the gate.

```
const taskStarting = running && !isRunning;  // the fix
...
if (running) {
  if (taskStarting) { _scrollLock = false; sb(); }
// The coverage gate used n < b - 1, excluding the LAST line of
// every marked region; n < b reports the true 11/11.
```

Figure 2: Task `12bbab41`: the fix and the gate defect, reconstructed from the task record.

We report test-blindness at the task level, and Section 4.5 treats the 34% broad label as an upper bound: on the 60-task re-coded sample, both readers mark 5 tasks, the second reader marks 7, and in 4 the second reader describes a defect reproduced after a green test suite. We keep this measure separate from finding counts. In cases like this one, simply asking the builder to write more tests may not help, because the builder's own verification system already reports success. Finding the problem requires an independent check of that verification system, including its coverage filters and other measurement logic, and the check should not rely on the system being checked. An independent reviewer can therefore provide a type of check that the builder's own verification cannot reliably provide.

F.2 REPAIR IS NOT ABSORBING: TWO ROUNDS TO BURY A DELETED KEY

Task `afc1c52d` shows why we measure *round-2 yield*. The task concerned a C++ key-value store that the builder had been optimizing in earlier tasks. An external audit had already found a serious problem in an earlier repair: making `Delete()` asynchronous caused a crash-resurrection bug. If `Delete()` returned and the process was killed before the tombstones reached durable storage, recovery could bring back about 60K of 97K deleted keys.

The builder's next repair added a delete-intent log. Each delete was recorded as a durable, 24-byte CRC-protected `[key|lsn]` record before `Delete()` returned. The complete local test suite passed, with 28 of 28 tests succeeding. The repair was still not safe. The first review round returned `REQUEST_CHANGES` with fourteen findings, including seven critical ones. These included recovery problems such as truncating the log before the matching sidecar data was durable, and log-only restarts that could skip replay and reuse sequence numbers.

The builder repaired all fourteen findings. However, this did not make the repair "absorbing": fixing the first set of problems did not simply make the system correct from that point onward. The second review round found seven *deeper* critical problems inside the new repairs. The clearest example was a no-fault linearizability race. An `Upsert` could publish a new value between the delete's mark and sweep phases. The sweep could then allow the old checkpointed value to appear again at runtime. Unlike the original crash-resurrection bug, this failure did not require a crash or restart.

The final repair moved the delete linearization point under the key's set lock. The second review then cleared the resulting lock-ordering concerns and confirmed that the scored hot path was unchanged. The final scored run achieved 5.50 Mops/s, the same as the pre-fix control.

The important point is that *every version of the repair passed the tests that existed when it was written*. These were therefore not simply cases where the builder ignored an already failing test. Each repair changed the system in ways that created or exposed new correctness requirements that the earlier tests and review had not covered. A successful test run, or even a successful review pass, can therefore give false confidence: the repaired system may pass its current checks while still containing problems introduced by the repair.

This task is a single case and gives no rate. Across the corpus, 60% of multi-round tasks with a determinate outcome (50.9% of all multi-round tasks) report at least one new finding in a later round, but Section 4.6 does not measure how many of those findings were caused by the preceding repair, and only 6 of the 160 sampled findings with a recorded source are later-round findings, too few to say. This task shows that the mechanism exists: a second review can find problems created by the first repair, which only become visible after the system has been changed. Here, the second round found seven additional critical defects without requiring further changes to the scored hot path. Both rounds ran as delegated subagents, so their cost is absent from the parent-side telemetry. The builder's own summaries give two different figures for it ("about $2.2, 0.2% of budget" for round one and "about 2% of budget" for both rounds), which is why Section 4.2 does not use narrated cost figures; we report no reviewer cost for this task.

F.3 A CODE-EXECUTION VULNERABILITY FOUND BY A ROUTINE REVIEW

Task `df411f40` initially looked like routine maintenance: migrate 636 catalog entries to a newer API by live-probing each model and updating a JSON catalog. The builder implemented the prober, updated the catalog, and extended an existing maintenance script. During this work, however, the script evaluated arithmetic expressions with Python's `eval()`, using an expression supplied by model output.

A reviewer, dispatched as a read-only subagent, identified this as its one critical finding among seven total findings. The problem was not simply the use of `eval()` in a maintenance script. It created a direct path from untrusted model output to code execution in a program running with the developer's privileges. Any model being probed, including a third-party model accessed through an aggregator endpoint, could potentially return an expression that the maintenance script would execute. The original task did not ask for a security review, so this vulnerability could easily have been missed by the builder's normal correctness checks.

The repair replaced `eval()` with a restricted evaluator based on Python's abstract syntax tree (AST). The expression is first parsed, and the evaluator accepts only numeric literals and a small list of arithmetic operators. Names, function calls, attribute access, and all other syntax are rejected. The builder also added regression tests with hostile expressions and checked that they were rejected. It then repaired the six remaining findings from the same review.

This case shows two points about independent review. First, the importance of a finding does not have to match the apparent importance of the task. Routine catalog maintenance contained a code-execution vulnerability because untrusted model output crossed a trust boundary into a privileged maintenance tool. Second, the finding came from an ordinary review. The reviewer was a routine delegated dispatch whose instruction set a $100 spending limit on a task with a $1,000 budget; the log records the limit but not the reviewer's actual spend, so we report no cost for this review either. Across the corpus, reviews also found critical findings in fifteen other tasks, and none of those reviews had been explicitly requested as a security audit. This suggests that independent review can

```
Fix 0 (earlier task)
  asynchronous Delete()
    -> Bug: crash-resurrection. SIGKILL before tombstones land;
       recovery resurrects ~60K of 97K deleted keys.

Fix 1
  delete-intent log: 24-byte CRC-protected [key|lsn] record
    appended before Delete() returns.
  Local suite: 28/28 pass.

Review round 1 -> REQUEST_CHANGES
  7 CRITICAL, 5 MAJOR, 2 MINOR
    C1  log truncated before sidecar durability
    C4  log-only restart skips replay and reuses LSNs
    (+12 more)

Fix 2
  all 14 findings repaired; truncation gated on a
  five-condition durability proof.

Review round 2 -> REQUEST_CHANGES
  7 deeper CRITICALs, including a no-fault linearizability race:
    an Upsert publishing between Delete's mark and sweep
    can cause the old checkpointed value to resurface.

Fix 3
  delete linearization moved under the key's set lock.

Re-examination
  lock ordering cleared; scored hot path confirmed untouched.
  Scored run: 5.50 Mops/s, identical to the pre-fix control.
```

Figure 3: Task `afc1c52d`, condensed from the task record. In this task the repair was not absorbing: each fix generation passed the tests available when it was written, yet the next review round uncovered additional critical defects inside the code the preceding repair had changed. The narrative for this one task attributes them to the repair; the corpus-wide share of such findings is not measured (Section 4.6). Round one found fourteen issues, including seven critical recovery flaws; after those were repaired, round two found seven deeper critical defects, including a no-fault linearizability race. The final repair preserved the scored hot path at 5.50 Mops/s, matching the pre-fix control. Both review rounds were delegated, so their cost is not measured.

```python
# update_models.py, as reviewed:
ctx = eval(expr)

# Repair recorded in the task: a restricted AST arithmetic evaluator.
tree = ast.parse(expr, mode="eval")
# Walk the tree; allow only numeric literals and + - * / // nodes.
# Reject Name, Call, Attribute, and every other node.
```

Figure 4: Task `df411f40`. The reviewer's critical finding was that the maintenance script passed model-derived input to Python's `eval()`, creating a code-execution path from untrusted model output to a script running with the developer's privileges. The repair parses the expression as an AST and permits only literal arithmetic operations, rejecting names, calls, attribute access, and other syntax. Regression tests exercise hostile expressions and assert rejection. The figure is a sketch based on the task record; the complete repair is provided in the released trace.

provide some security coverage as part of ordinary correctness review, even when security is not part of the task specification.