METRICS-IN-THE-LOOP: CHEAP, TIMELY COURSE CORRECTIONS IMPROVE CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

In spite of their impressive recent progress, coding agents still accumulate overly complex functions and untested code over long trajectories. Nothing flags these weaknesses as they form: the only signal is a failing test, and by then they likely have compounded. We introduce metrics-in-the-loop, an inference-time intervention that gives agents in-flight feedback: cheap, automatically computed code metrics injected into the agent's context during the trajectory. After each code-changing agent turn, we insert short feedback messages naming changed functions whose cyclomatic complexity exceeds a limit (complexity feedback) or changed lines that are skipped by tests (coverage feedback), giving the agent the opportunity to refactor code or strengthen tests before continuing. On two Python benchmarks, complexity feedback consistently leads agents to reduce the concentration of codebase complexity. It does so with no detrimental impact on functional correctness. Generic reminder messages, on the other hand, have almost no effect. Our experiments investigate a practical form of agent "re-railing", where timely, actionable measurements can effectively redirect work before local weaknesses become embedded, suggesting practical paths for coding agent to work effectively on long-term-maintainable codebases.

1 INTRODUCTION

Software engineering involves more than writing code that works: code must also be organized so that it remains testable and easy to change as requirements evolve (Parnas, 1972; Lehman, 1980). Coding agents now write code faster than any human can review, and how agents write often departs from well-engineered human code, favoring large, monolithic, single-file implementations (Yang et al., 2026). The problem compounds when an agent builds on its own work: successive extensions concentrate complexity and duplicate code (Orlanski et al., 2026), and evolving software across many files without breaking what already works remains difficult (Le et al., 2025). As tasks become longer-horizon and agent-written codebases extend into millions of lines, increases in code complexity increase the likelihood of introducing bugs, inefficiencies and vulnerabilities, making ongoing maintance progressively more difficult.

Yet as it writes code, an agent typically receives very little help: review steps, whether by humans or machines, typically happen only after the agent has finished. Unit tests do give feedback during generation, but they mostly check functional correctness: a passing suite does not show which functions have become overly complex, nor which newly written lines are missed by tests. Task instructions to the agent do not close the gap either: on one of the benchmarks we study, SlopCodeBench (Orlanski et al., 2026), providing quality guidance in the prompt improves the initial implementation, but does not prevent later degradation. Task instructions merely offer general guidance, not specific suggested improvements. Problems that could be fixed cheaply while the code is being written are therefore left to a later, separate pass, if they are fixed at all.

We ask whether cheap, mechanical feedback delivered during code generation can improve code structure without sacrificing functional correctness. We introduce **MITL** (**M**etrics-**I**n-**T**he-**L**oop), a lightweight plugin that sits between a coding agent and its workspace. After each turn that changes code, MITL measures what changed and returns a short report before the agent continues. The report draws on two standard signals from software engineering. *Cyclomatic complexity* (McCabe, 1976) counts the independent paths through a function; MITL flags each changed function above a given

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.

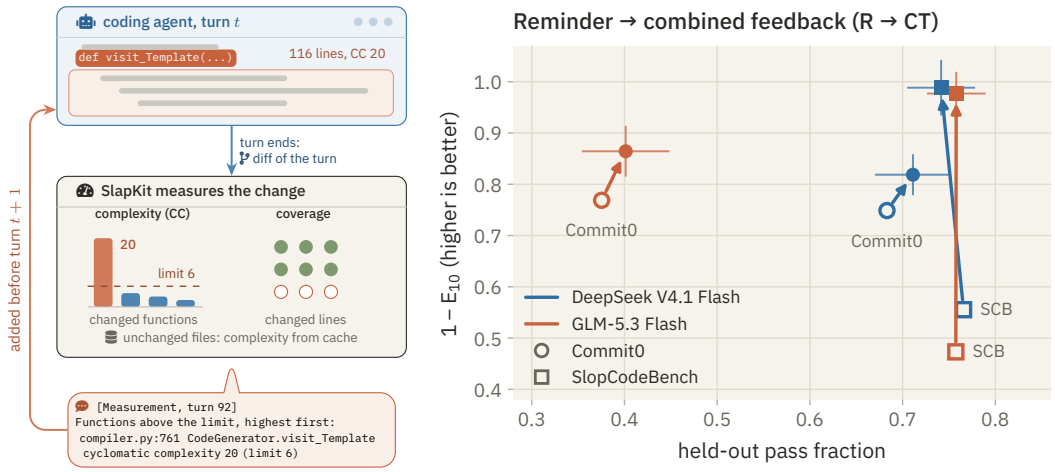

Figure 1: Metrics-in-the-loop. **(a)** One round of the loop: when an agent turn changes code, we measure the new cyclomatic complexity of the changed functions and which changed lines existing tests now execute (middle box); a specific feedback message to the agent enters the agent's context before the next turn. **(b)** Adding cyclomatic complexity and test coverage feedback to the reminder lowers erosion, a measure of complexity concentration across the codebase, on our studied benchmarks while preserving functional correctness. **Axes:** held-out pass fraction and $1 - E_{10}$, *higher is better on both.*

limit (6 in our experiments). *Test coverage* records which lines a test suite executes (Inozemtseva & Holmes, 2014); MITL flags changed lines that current tests do not reach. Both measurements come from SlapKit, a function-level measurement tool (Appendix B). MITL requires no training, no reference solution and no access to held-out tests, and it attaches to an existing agent without modifying the model. Because it reports on code the agent is still working on, the agent can simplify a function or add a test while the relevant context is fresh. Figure 1(a) shows one round of this loop.

Feedback messages to the agent, called reports, do not just provide metrics: they also include reminders to keep code simple and well tested. Simple metric-free feedback, just by being provided often, could also change the agent's behavior. We therefore carefully compare three measurement arms (providing, respectively, complexity (C), test coverage (T), or both kinds (CT) of feedback) with a reminder control (R), which receives the same review instructions on the same schedule but without any measurement, and with a baseline (B0) receiving no messages at all (Section 3). We evaluate on two benchmarks that capture different stages of a codebase's life. In Commit0 (Zhao et al., 2025), the agent reconstructs a Python library from its specification. We use 25 libraries for which we withhold the official test suite in order to force the agent to write its own tests, and the held-out suite serves only to measure correctness. In SlopCodeBench (Orlanski et al., 2026), the agent builds a program over three to eight checkpoints, each introducing specification changes that extend the previous one. The agent starts each checkpoint with a fresh context but keeps its earlier code, so structural choices made early carry over to later work. We test two leading open-source coding models, DeepSeek V4.1 Flash and GLM-5.3 Flash, under a fixed token budget.

Our results show that complexity feedback reduces *structural erosion*, defined below, a measure of how complexity accumulates in a few functions across a codebase. Feedback arms that include complexity (C and CT) show an extremely significant drop in erosion under all experimental conditions. Complementary measures, such as verbosity, not provided as explicit feedback, also move favourably, along with hidden tests covering 2–5 percentage points (pp) more of the agent's code.

Meanwhile, held-out correctness does not change appreciably. On Commit0, combined feedback raises the pass fraction by 4.4–4.7 points with intervals that (barely) cover zero; on SlopCodeBench the change is -2.4 and 0.0 points (intervals cover zero). Coverage feedback makes agents raise the coverage of the functions they edit on SlopCodeBench but not on Commit0, and it does not reduce erosion; the reminder alone does so only on SlopCodeBench, by less than half as much as the complexity report. Figure 1(b) summarizes both experiments for the combined feedback, and complexity feedback alone behaves the same way (Appendix I). We make the following contributions:

1. **Local measurement interventions for repository-level coding.** Metrics-in-the-loop returns advisory reports on the code an agent is currently changing, including complexity and test coverage. It requires neither model training nor access to reference solutions or held-out tests.
2. **Controlled experiments.** We rigorously assess contributions of each kind of feedback through five experimental arms, which include a "no-feedback" baseline (B0) and a "trivial feedback" placebo, on two agentic coding benchmarks, assessing correctness with withheld tests.
3. **Trace-level evidence that feedback changes agent behavior.** Under two models on both benchmarks, complexity feedback significantly lowers the concentration of codebase complexity, and agents rework the functions it names; held-out correctness does not change detectably, and coverage feedback alone does not lower erosion. We audit both agent execution traces and final code to separate genuine decomposition from surface-only cosmetic changes.

2 Related work

Feedback during generation. Inference-time feedback is an established way to improve language-model outputs. Self-Refine alternates generation, model-produced feedback and revision (Madaan et al., 2023), Reflexion retains verbal reflections across trials (Shinn et al., 2023), and self-debugging draws on execution results (Chen et al., 2023). Fine-Grained RLHF shows the value of feedback that locates where an output falls short, rather than restating what is wanted (Wu et al., 2023). Our reports are external measurements of the current implementation, injected during repository work, and we test whether they add value beyond the same instruction delivered on the same schedule, not whether feedback helps in general.

Complexity and coverage as feedback. The closest antecedent is Sepidband et al. (2025), who use complexity metrics to guide regeneration of incorrect code. Their procedure studies a broad set of metrics, selects informative ones using correctness labels, and includes an extension that adds complexity feedback to Reflexion. Complexity feedback has thus been tested before, including alongside execution feedback, with comparatively modest gains over that iterative baseline. We ask a narrower question in a different setting: whether a fixed, local complexity limit can steer an agent throughout the construction and extension of a codebase, and how it interacts with coverage feedback. We measure the structural response and held-out correctness separately, because an agent can comply with a structural request without solving more of the task. Coverage, for its part, is an established input to test generation (Altmayer Pizzorno & Berger, 2025; Alshahwan et al., 2024), and agent interfaces already give edit-time feedback such as syntax checks (Yang et al., 2024). In our setting the agent changes its implementation as well as its tests: its own tests drive the feedback, and a withheld suite judges correctness.

What the metrics establish. Cyclomatic complexity is related to, but not interchangeable with, code size (Landman et al., 2016), and neither individual metrics nor their combinations capture understandability well (Scalabrino et al., 2021); we are careful not to equate drops in complexity with increased maintainability. Before language models, complexity was repeatedly associated with maintenance cost (Lehman, 1980; Banker et al., 1993), but proactive reductions were rarely tested causally, and the evidence that exists is mixed once code size is controlled (Sjøberg et al., 2013; Gil & Lalouche, 2017). For coding agents, cleaner starting code lowers token use without raising pass rates (Trivedi & Schmitt, 2026); we intervene instead while the code is being written.

Benchmarks. Commit0 evaluates the reconstruction of complete libraries (Zhao et al., 2025); we withhold its official tests. SlopCodeBench evaluates repeated extensions of an agent's own program and measures structural erosion (Orlanski et al., 2026), and SWE-Milestone likewise studies dependent stages of software evolution (Deng et al., 2026). Because we modify both benchmarks' harnesses, our absolute scores should not be read as leaderboard comparisons. Appendix A discusses further related works.

3 Metrics in the loop

3.1 Local measurements during a coding trajectory

The agent works in turns, one model step each. After every turn that changes a file, the harness that runs the agent snapshots the workspace, measures the functions the turn changed, and places a

Table 1: Feedback arms. Every arm except B0 receives the same reminder under the same delivery rule; CT shows the two tables separately.

Arm	Information delivered during the trajectory
B0	No feedback block
R	Reminder only
C	Reminder and complexity table
T	Reminder and uncovered-changed-lines table
CT	Reminder and both tables

short message in the agent's context before its next model call; details are provided in Appendix B. Formally, let x_t denote the workspace after turn t, and D_t the scored functions that the turn's diff touches; a message follows turn t when x_t can be measured and D_t is not empty. If no function at a turn exceeds limits, an empty table is produced; a turn that cannot be measured (e.g. due to unparseable code) produces no message, rather than one that suggests the code is clean.

The *complexity report* lists the functions in D_t whose cyclomatic complexity (McCabe, 1976) exceeds a given limit (set to 6 in our experiments, from common rules-of-thumb in software engineering), highest first, and counts those at or under it. Likewise, the *coverage report* lists the functions in D_t whose added or edited lines include some that existing tests do not cover; we target having zero such lines. They answer different questions: respectively on the concentration of control flow and whether a changed line has run at all. Neither measures semantic correctness.

3.2 Interventions and Controls

The four arms R, C, T and CT form a 2×2 design in the presence of complexity and coverage information (Table 1); B0 receives no messages. The review instructions that end every message are byte-identical across the four, so CT $-$ R estimates the effect of adding both reports to the reminder, while T $-$ R and T $-$ R isolate each. Measurement arms also differ from R in the message header and in one paragraph of the task prompt, which announces the messages and defines each metric and its target (Appendix C). As we do not want to fix the number of reports ahead of time, an intervention can change what the agent edits, how often its code can be measured and when it stops; under a common budget, these differences are part of its effect. We therefore report the realized number of measured turns (Table 6) rather than condition the primary estimate on it.

We leave out two alternatives by design: CT provides feedback as the two tables shown consecutively, rather than a composite score such as CRAP, which folds complexity and coverage into one number (Savoia, 2007), because the experiment tests the two signals separately. And every report arrives while the agent is still working, before its next turn. The design compares this with a reminder and with no message, but not with the same reports delivered once the agent has finished, so it tests the policy of measuring and reporting during the trajectory as a whole and cannot say how much of the effect is due to the timing rather than the content of the report.

3.3 Outcomes and Estimands

We define held-out correctness as the fraction of a fixed set of held-out tests that agent-generated code passes: Commit0's official suite, withheld from the agent, or SlopCodeBench's hidden tests for the final checkpoint. It is separate from the agent's own generated test results and from the coverage shown in its context. Since test cases within a task are not independent, our unit of analysis is entire tasks: we first average the available seeds within each task (a Commit0 library or a SlopCodeBench problem), then average paired arm differences across tasks:

$$\widehat{\Delta}_{\mathrm{CT-R}} = \frac{1}{|\mathcal{L}|} \sum_{\ell \in \mathcal{L}} \left(\overline{Y}_{\ell,\mathrm{CT}} - \overline{Y}_{\ell,\mathrm{R}} \right), \tag{1}$$

where $\mathcal{L}$ contains the tasks with readings in both arms, each weighted equally. The primary estimate is reported separately for each model, with a paired t interval across tasks and a task bootstrap as a sensitivity analysis.

Table 2: The two experiments. Both run the five arms of Table 1 with the same feedback wording under DeepSeek V4.1 Flash and GLM-5.3 Flash.

	Commit0	SlopCodeBench
Task	reconstruct a Python library from its specification	build a program over 3–8 checkpoints, each a specification that extends the previous one
Tasks	25 libraries	35 problems
Cell	one library under one arm, model and seed	one problem's whole checkpoint sequence under one arm, model and seed
Seeds	3; 5 for R and CT	2
Cells launched / analyzed	950 / 909	700 / 640
Budget	2M processed tokens per cell	1M processed tokens per checkpoint
Context	one conversation	fresh at each checkpoint; workspace, own tests and notes kept; every checkpoint runs even if the previous one failed
Correctness	pass fraction on the withheld official suite, at the last graded snapshot at or below the cap	strict pass fraction on the final checkpoint's hidden tests, regression tests included

To measure how concentrated a codebase's complexity is, we adapt the erosion index of SlopCodeBench (Orlanski et al., 2026). For each scored production function f, let c_f be its cyclomatic complexity and n_f its number of nonblank, noncomment source lines. Define

$$E_\tau(x) = \frac{\sum_f c_f \sqrt{n_f}\, \mathbf{1}\{c_f > \tau\}}{\sum_f c_f \sqrt{n_f}}. \tag{2}$$

E_τ is the share of complexity mass, with each function's complexity weighted by the square root of its length, held in functions above τ. We report E_{10}, echoing SlopCodeBench's threshold of 10. (Appendix F reports E_6 and the share of functions over the limit as a manipulation check.) A smaller erosion value means that the complexity mass is spread more evenly across the codebase, and less of it lies in functions above the threshold; it does not, by itself, establish greater maintainability or correctness. We compute two versions of erosion: (i) the primary one, restricted to the agent's production code, and (ii) SlopCodeBench's version, computed by the benchmark's own checker over every file in the snapshot, the agent's tests included; Section 5.1 reports both, and Appendix G tabulates the benchmark's measures with intervals. SlopCodeBench also defines a *verbosity* score: the share of a program's source lines that the benchmark's checker either flags with one of its 137 syntax-pattern rules for verbose constructions and commonly cited anti-patterns, or identifies as structural clones of other lines, a line counting once however many rules hit it (Orlanski et al., 2026, Eq. 4). We compute it with the benchmark's pinned checker (`scb-check 0.1.3`) on the same snapshots. We emphasize that the agent is never given feedback that include the benchmark's erosion score or verbosity.

4 Experimental setup

4.1 Benchmarks, models and budgets

Table 2 summarizes the two benchmarks. Commit0 (Zhao et al., 2025) asks the agent to reconstruct a library from its specification; the official tests are withheld from the intended agent interface, so the agent writes its own, and the withheld suite is used to measure correctness. SlopCodeBench (Orlanski et al., 2026) asks it to build a program over three to eight checkpoints, each introducing a specification that extends the previous one, so that structural choices made early carry into later work. Both experiments run the five arms of Table 1 with the same feedback wording under DeepSeek V4.1 Flash and GLM-5.3 Flash, configured for high reasoning effort. Every cell runs the same coding agent, Oh-My-Pi 18.1.22 (Bölük, 2026), headless and with the same tool set (Appendix B). (A smaller exploratory run with Qwen 3.8 27B on a SlopCodeBench subset is in Appendix M.) Processed tokens count uncached input, completion including reasoning, and cache writes, but not cache reads. On both benchmarks, CT $-$ R on held-out correctness is the primary contrast and E_{10} at the endpoint is secondary.

Table 3: Change in E_{10} (share of complexity mass in functions with CC > 10; lower is better) at the final tree. Paired 95% t intervals: Commit0 pairs arm means within library (24–25 libraries; seeds 1–5 for R and CT, 1–3 for B0, C, T); SCB pairs arm means within problem (32–35 problems, 2 seeds). Cells with no measured E_{10} are dropped (complete case).

Benchmark	Model	C–R	CT–R	T–R	R–B0
Commit0	DeepSeek	-0.123 [-0.185, -0.061]	-0.070 [-0.110, -0.030]	$+0.007$ [-0.009, +0.024]	-0.002 [-0.019, +0.015]
Commit0	GLM	-0.120 [-0.177, -0.064]	-0.096 [-0.145, -0.046]	$+0.015$ [-0.020, +0.049]	-0.021 [-0.056, +0.014]
SCB	DeepSeek	-0.428 [-0.483, -0.372]	-0.433 [-0.487, -0.379]	$+0.067$ [+0.030, +0.104]	-0.202 [-0.242, -0.162]
SCB	GLM	-0.529 [-0.573, -0.485]	-0.504 [-0.546, -0.461]	$+0.056$ [+0.027, +0.085]	-0.180 [-0.210, -0.149]

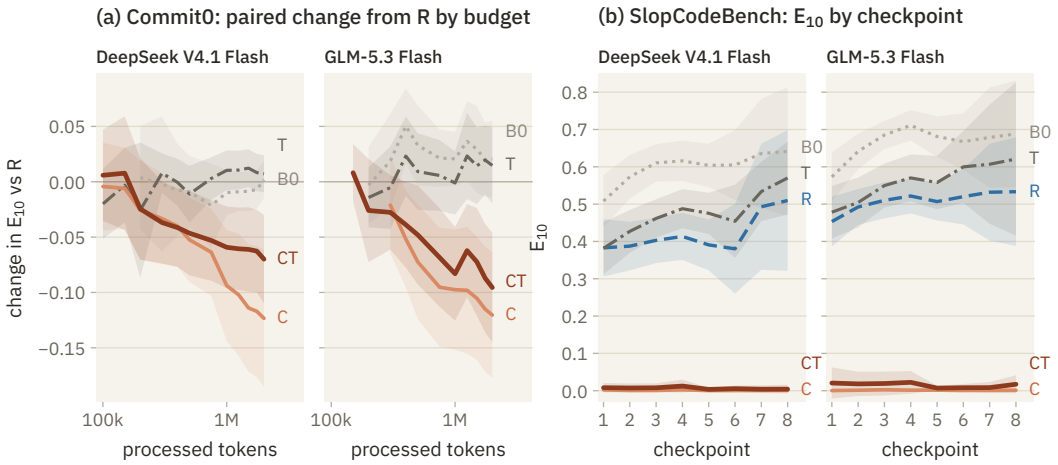

Figure 2: The erosion gap opens within a run and persists across extensions. **(a)** Commit0: paired change in E_{10} relative to R at each processed-token budget (log-scale x-axis, up to 2M limit), library-paired, with 95% t bands; **(b)** SlopCodeBench: E_{10} at each checkpoint by arm, mean over problems of the problem mean with a 95% t band; not all problems extend to 8 checkpoints.

4.2 Analysis population

Commit0 ran at least three seeds per arm and model (exact conditions in the results tables). A graded workspace on which the hidden suite cannot import or complete scores zero on the whole fixed set of held-out tests (150 cells: 106 GLM, 44 DeepSeek); our primary analysis keeps these zeros, and Appendix Table 13 reports estimates without them. SlopCodeBench ran two seeds per arm and model on 35 of the benchmark's 36 problems (one, datagate, was excluded before any cell ran because the coverage step could not measure its generated files); a cell that failed for infrastructure reasons was rerun once.

Integrity. We carried out an LLM (Opus 5.5) audit of every trace: within the limits of automated LLM critiques, we found that no SlopCodeBench agent reached anything outside its workspace: its operating-system fence blocked every attempt. On Commit0 a path-word filter was routinely bypassed, and the 28 cells that reached hidden or finished-library material are set aside (Appendix H).

5 Results

Tables 3 and 5 give the four contrasts of Section 3 on both benchmarks, paired within task and reported per model. We now detail these findings.

5.1 Complexity feedback lowers erosion on both benchmarks

Relative to the reminder alone, complexity feedback lowers E_{10} at the endpoint on both benchmarks and under both models, with every 95% confidence interval excluding zero (Table 3). On Commit0, C $-$ R is -0.123 under DeepSeek and -0.120 under GLM, and CT $-$ R respectively -0.070 and

Table 4: Response to feedback (included cells; Commit0 seeds 1–3). Complexity: share of over-limit functions edited again / share under the limit at the end of the run. For C and CT these are the functions the table named; for R, over-limit functions the agent edited, which nobody named. Coverage: of the functions an agent edited while their coverage was below 1, the share whose coverage later rose; the same population in every arm, so the arms are comparable. From the response audit.

		complexity: edited / under limit			coverage rose		
Benchmark	Model	R	C	CT	R	T	CT
Commit0	DeepSeek	20% / 1%	54% / 46%	36% / 23%	64%	57%	57%
Commit0	GLM	30% / 3%	60% / 58%	46% / 35%	40%	42%	34%
SCB	DeepSeek	38% / 5%	94% / 94%	84% / 82%	17%	95%	92%
SCB	GLM	41% / 4%	90% / 94%	82% / 85%	10%	38%	63%

-0.096. On SlopCodeBench the changes are four times larger: C − R is -0.428 and -0.529, and CT − R is -0.433 and -0.504. The difference is not only one of starting point. Under the reminder alone, about a quarter of Commit0's complexity mass sits in functions above 10 (0.25 and 0.23), against 0.43 and 0.53 on SlopCodeBench; complexity feedback removes about half of that share on Commit0 and nearly all of it on SlopCodeBench.

Figure 2 shows when the gap forms. On Commit0 it opens steadily as the budget is spent rather than appearing at the end. On SlopCodeBench it is present from the first checkpoint and holds through the context reset at every later one: under R the share stays between 0.38 and 0.53 across checkpoints, under C at or below 0.003 and under CT at or below 0.022: complexity feedback keep erosion near zero at every stage of the program's growth rather than repairing it at the end. Measures the agent is never shown move with it on SlopCodeBench (Appendix Table 10): relative to the reminder, complexity feedback lowers the benchmark's own erosion score by 0.26–0.29 and its verbosity score (Section 3.3) by 3.4–4.5 points, and the hidden tests execute 2.4–5.1 points more of the agent's code, every interval excluding zero. The reminder alone already moves all three relative to no message (erosion score -0.16, verbosity -3.5 to -3.6 points, hidden-suite coverage $+2.9$ to $+3.3$ points), and complexity feedback moves each of them again on top of it, by a similar amount for verbosity and coverage and by nearly twice as much for the erosion score. On Commit0, where only hidden-suite coverage can be measured, no feedback contrast resolves. A statistical mixed-effects analysis also agrees on both benchmarks (Appendix Table 11, along with sensitivity variant rules in Table 13).

5.2 Agents do rework according to the feedback they receive

This erosion drop reflects agents acting on the feedback they receive. In every trace, we audited each over-limit or under-covered function that the agent edited, regardless of whether was named in feedback. Of the over-limit functions named in arm C, agents edit again 54–60% on Commit0 and 90–94% on SlopCodeBench, at a respective a median of two and three turns later; of those, a respective 46–58% and 94% end the run under the limit, against 1–5% of the over-limit functions the agent edits under R (Table 4). The response is weaker under CT than under C on both benchmarks, as if the second feedback (test coverage) competed for the agent's attention. Coverage feedback, on the other hand, is mostly effective within SlopCodeBench: of the functions an agent edited while their coverage was below 1, coverage later rises far more often under T and CT than under R there, and no more often on Commit0. On Commit0, coverage feedback changes neither coverage nor structure. Additional details appear in Appendix L.

5.3 Understanding structural decomposition after complexity feedback

Splitting a function mechanically lowers its contribution to the erosion score (2), whether or not the split improves the design. On both benchmarks most of the reduction comes from extracting single-use helpers: undoing every single-use extraction removes almost all of it on Commit0 (for DeepSeek under C, from -0.13 to -0.02) and leaves a reduction of only 0.04–0.10 on SlopCodeBench. Single-use does not mean cosmetic, although some of it might be. A structural audit of the final trees finds that on Commit0, C and CT carry no more pass-through forwarders than R and the same share of trivial functions as R and as the libraries' human-written originals. On SlopCodeBench, C and

Table 5: Change in held-out pass fraction, percentage points (Commit0: at the 2M-token cap; SCB: strict hidden-test pass at the final scheduled checkpoint). Paired 95% t intervals as in Table 3; Commit0 uses seeds 1–5 for R and CT and 1–3 for the other arms. Commit0 CT–R is the pre-registered primary contrast; Table 8 separates the pre-registered three seeds from the five-seed estimate.

Benchmark	Model	C–R	CT–R	T–R	R–B0
Commit0	DeepSeek	$+3.3$ [-0.5, +7.2]	$+2.8$ [-1.3, +6.9]	$+1.2$ [-3.3, +5.6]	$+5.3$ [-3.0, +13.6]
Commit0	GLM	-0.2 [-6.1, +5.6]	$+2.6$ [-2.1, +7.3]	-0.6 [-6.7, +5.4]	$+0.8$ [-4.7, +6.3]
SCB	DeepSeek	$+1.9$ [-0.7, +4.5]	-2.4 [-6.1, +1.2]	-0.1 [-2.7, +2.5]	-0.8 [-4.5, +2.8]
SCB	GLM	-1.4 [-4.2, +1.4]	$+0.0$ [-3.1, +3.2]	$+2.1$ [-3.8, +8.0]	-1.1 [-5.4, +3.3]

CT programs carry about four to eight more trivial single-use helpers each, deeper call chains and more dispatch tables than R. Total branching is unchanged on Commit0 and falls by roughly 15–20% on average on SlopCodeBench, and in a hand-read sample of the 20 largest drops across both benchmarks, 17 split the code into named, testable concepts, 2 were mixed and 1 was cosmetic. On Commit0, complexity feedback thus spreads the same control flow over more, smaller functions rather than removing it altogether; on SlopCodeBench it also removes some.

We read twelve Commit0 cases turn by turn to better understand the effect of feedback (Appendix O, which also shows representative decompositions and wrappers from both benchmarks). Two patterns stand out. First, a fix can make complexity worse. We witnessed that in some substantial decompositions, new helpers may be written that are themselves over the limit; yet in spite of receiving feedback, the agent decides to never edits them again and they remain over the limit until the last turn: this is because feedback only covers functions that are changed in the current turn and "forgotten" functions might never be addressed. Second, coverage feedback elicits a targeted test but not always a complete one: flagged lines sometimes become fully covered, sometimes only partly so.

5.4 Held-out correctness does not change detectably

There are no feedback arm that changes held-out correctness statistically significantly on either benchmark (Table 5). On Commit0, in the three-seed design, combined feedback raised the held-out pass fraction relative to R by 4.43 points under DeepSeek and 4.73 under GLM, but both intervals include zero, so neither model meets the criterion for a benefit (Appendix Table 8). With seeds 4 and 5 added, the estimates become 2.80 and 2.60 points. A post hoc inverse-variance pool of the two models gives 4.56 points [0.59, 8.53] at three seeds and 2.71 [-0.22, 5.65] at five, and a binomial mixed-effects model (Appendix J) supports the same conclusion (CT–R odds ratio 1.46 [0.88, 2.41] at three seeds, 1.28 [0.87, 1.88] at five). The estimate is also sensitive to the repositories that scored zero because the hidden suite produced no report: without them, the pooled five-seed estimate is 0.1 points (Table 13). Together, these results suggest that the effect of feedback on functional correctness for Commit0 is noise.

On SlopCodeBench, the strict pass fraction is high (0.76–0.78 on average at the final checkpoint) and agents stop on their own at most checkpoints, well under the 2M token cap. CT–R is -2.4 points under DeepSeek and 0.0 under GLM, and C–R 1.9 and -1.4 (Table 5); this is supported by the binomial mixed-effects model, with a CT–R odds ratio of 0.91 [0.77, 1.07] across both models and C–R of 1.01 [0.86, 1.19]. It is likely that the relative saturation of the held-out tests under the baseline leave little room for feedback arms to exhibit meaningful effect.

5.5 Coverage feedback and the reminder alone

We visually present additional contrasts (C–R, T–R, R–B0) in Appendix I, similar to Figure 1(b). Coverage feedback alone does not lower erosion: T–R spans zero on Commit0 and raises E_{10} slightly on SlopCodeBench, by 0.06–0.07. The reminder alone lowers erosion only on SlopCodeBench, by 0.18–0.20 (R–B0), less than half of what complexity feedback removes; on Commit0 it leaves E_{10} where it was. Neither changes correctness detectably: every T–R and R–B0 pass-fraction interval spans zero, on both benchmarks. We obtain results in broad agreement using a formal mixed-effects statistical model (Table 11). Appendix B reports the measurement overhead, and Appendix N an exploratory analysis of token and step use on Commit0: relative to R, neither C nor CT uses fewer tokens or steps under either model.

6 Discussion and outlook

A short measurement of the code an agent has just changed, delivered before its next turn, changes how the agent structures that code. The effect holds on both benchmarks and under both models, with no significant degradation in held-out correctness, and it comes cheaply: complexity feedback needs a static parse of the files changed after an agent turn, lends itself well to caching, with a message size of one to two kilobytes, under 1.1% of the tokens a cell processes (Appendix B). Coverage feedback, which reruns the agent's own suite after every measured turn, is more expensive, yet did not lower erosion.

Jointly with our primary code erosion measure, we also see improvement in metrics that the agent never sees (Appendix G). On SlopCodeBench the benchmark's own erosion score, from a different tool over a broader corpus, falls by 26–29 percentage points; its code carries fewer dublicated or verbose lines, with verbosity score falling by 3.4–4.5 points; and withheld test coverage improves by 2.4–5.1 points. These are all markers that the software engineering community associates with maintainability and readability: and they are all positively impacted by a local, mechanical yet specific signal, provided in a timely fashion to the coding agent, something that a generic reminder does not achieve.

Whether these markers make a codebase cheaper to maintain has long remained an open question in software engineering: while complexity, for decades, has been known to be associated with maintenance cost, controlled studies of it have been almost entirely lacking (Appendix A). Coding agents remove that obstacle. The same specification can be built many times, with and without complexity control, and the resulting trees can receive the same sequence of held-out change requests. Such a design enables measuring the cost of each later change in tokens and turns, the regressions the hidden tests catch, non-functional properties such as performance and security, and whether code that starts out more modular churns less, a saving that is not witnessed within one task alone (Appendix N). Nor need the control stop at cyclomatic complexity: additional metrics such as the CRAP score that joins complexity and coverage (Savoia, 2007), duplication, coupling between modules and forms of mutation testing are all measurable after every turn.

It will be necessary to invest in building and cultivating better benchmarks. Current near-frontier LLMs have trained on essentially all public code; Commit0's libraries and SlopCodeBench's specifications are public, and both models we ran postdate them, so contamination cannot be excluded, however carefully hidden material is fenced off (Appendix H). Reconstructing a known library is the more exposed of our two tasks, extending a fresh program from a specification the less. Future studies will need longitudinal measurements on closed-source repositories, where neither the code the agent extends nor the tests that judge it have been in any training set.

Several limits remain. The evidence covers Python, one agent harness, two models, and one token budget per benchmark. The design does not vary when a report is delivered, so it cannot identify a schedule. We also keep the complexity threshold fixed at six.

Finally, our feedback intervention can be understood as a harness change, a form of *harness engineering*. Component-level ablations show that planning, tool interfaces and context management each move a coding agent's success rate by several points, in directions that depend on the model (Fan et al., 2026), and closed loops that let an agent evolve its own harness already lift pass rates beyond hand-built designs (Lin et al., 2026). Metrics-in-the-loop is one such component, cheap to add. The same loop could carry other measurements, adapt its limit to the model, and learn when to express feedback. Much still has to be understood about how all these components interact. We inherited our notions of clean code from people, and we do not know whether the same notions should apply to readers with context windows, or which properties of code at one time predict where and how an agent will fail at a later one. But as coding agents become entrusted with writing and maintaining a growing fraction of all software, this has become an experimental question worth answering.

AI-USE STATEMENT

Generative AI tools assisted with software implementation, data analysis, literature discovery, and manuscript drafting and editing, but did not replace humans. The latter are responsible for the entire contents of this paper.

REPRODUCIBILITY STATEMENT

Section 3 defines the arms, outcomes and estimands, and Section 4 the two experiments and their analysis populations. Appendix B describes how measurements are taken and delivered, Appendix C reproduces every prompt and message, and Appendices J and K give the supporting models and sensitivity rules.

REFERENCES

Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024. URL https://arxiv.org/abs/2402.09171.

Juan Altmayer Pizzorno and Emery D. Berger. CoverUp: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2025. doi: 10.1145/3729398. URL https://arxiv.org/abs/2403.16218v4.

Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, 2009. doi: 10.1109/TSE.2008.89.

Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, 1993. doi: 10.1145/163359.163375.

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier de Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. WorkArena++: Towards Compositional Planning and Reasoning-based Common Knowledge Work Tasks. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*, 2024. doi: 10.52202/079017-0195. URL https://arxiv.org/abs/2407.05291.

Can Bölük. Oh-My-Pi: Coding agent with the IDE wired in. Software, version 18.1.22, released 2026-09-14, 2026. URL https://omp.sh. Source: https://github.com/can1357/oh-my-pi.

Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 465–475, 2017. doi: 10.1145/3106237.3106259.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug, 2023. URL https://arxiv.org/abs/2304.05128.

Bill Curtis, Sylvia B. Sheppard, Phil Milliman, M. A. Borst, and Tom Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering*, SE-5(2):96–104, 1979. doi: 10.1109/TSE.1979.234165.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. SWE-Milestone: Evaluating AI Agents on Continuous Software Evolution, 2026. URL https://arxiv.org/abs/2603.13428v4.

Atish Kumar Dipongkor, Talank Baral, Wing Lam, and Kevin Moran. Test Coverage Analysis of Agentic Pull Requests, 2026. URL https://arxiv.org/abs/2607.18057.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vázquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How Capable are Web Agents at Solving Common Knowledge Work Tasks? In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001. doi: 10.1109/32.895984.

Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27 (7):630–650, 2001. doi: 10.1109/32.935855.

Amine El Hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. WebArena Verified: Reliable Evaluation for Web Agents. In *NeurIPS 2025 Workshop on Scaling Environments for Agents (SEA)*, 2025. URL https://openreview.net/forum?id=94tlGxmqkN.

Amine El Hattami, Nicolas Chapados, and Christopher Pal. SKILL.nb: Selective Formalization and Gated Execution for Durable Agent Workflows. In *ICML 2026 Workshop on Failure Modes in Agentic AI (FAGEN)*, 2026. URL https://openreview.net/forum?id=zYh5EVYEtq.

Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, and Xiaoyang Wang. An empirical study of harness design for coding agents, 2026. URL https://arxiv.org/abs/2609.20804.

Yossi Gil and Gal Lalouche. On the correlation between size and metric validity. *Empirical Software Engineering*, 22(5):2585–2611, 2017. doi: 10.1007/s10664-017-9513-5.

Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, 1991. doi: 10.1109/32.106988.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, 2014. doi: 10.1145/2568225.2568271. URL https://doi.org/10.1145/2568225.2568271.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 654–665, 2014. doi: 10.1145/2635868.2635929. URL https://doi.org/10.1145/2635868.2635929.

Chris F. Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, 1995. doi: 10.1007/BF02249043.

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014. doi: 10.1109/TSE.2014.2318734.

Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process*, 28(7):589–618, 2016. doi: 10.1002/smr.1760. URL https://doi.org/10.1002/smr.1760.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. The BrowserGym Ecosystem for Web Agent Research. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=5298fKGmv3.

Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. doi: 10.1109/PROC.1980.11805.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026. URL https://arxiv.org/abs/2604.25850.

Ximing Lu, Faeze Brahman, Peter West, Jaehun Jang, Khyathi Chandu, Abhilasha Ravichander, Lianhui Qin, Prithviraj Ammanabrolu, Liwei Jiang, Sahana Ramnath, Nouha Dziri, Jillian Fisher, Bill Yuchen Lin, Skyler Hallinan, Xiang Ren, Sean Welleck, and Yejin Choi. Inference-time policy adapters (IPA): Tailoring extreme-scale LMs without fine-tuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://arxiv.org/abs/2305.15065.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.17651.

Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4): 308–320, 1976. doi: 10.1109/TSE.1976.233837. URL https://doi.org/10.1109/TSE.1976.233837.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: Benchmarking how coding agents degrade over long-horizon iterative tasks, 2026. URL https://arxiv.org/abs/2603.24755v2. Version 2.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018. doi: 10.1007/s10664-017-9535-z.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pp. 279–287, 1994. doi: 10.1109/ICSE.1994.296790.

Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 367–377, 2009. doi: 10.1109/ESEM.2009.5314233.

Alberto Savoia. Pardon my french, but this code is C.R.A.P. (2). Artima Developer weblog, July 2007. URL https://www.artima.com/weblogs/viewpost.jsp?thread=210575.

Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability. *IEEE Transactions on Software Engineering*, 2021. doi: 10.1109/TSE.2019.2901468. URL https://doi.org/10.1109/TSE.2019.2901468.

David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, 2011. doi: 10.1109/ICST.2011.32. URL https://doi.org/10.1109/ICST.2011.32.

Melika Sepidband, Hamed Taherkhani, Song Wang, and Hadi Hemmati. Enhancing LLM-based code generation with complexity metrics: A feedback-driven approach. In *IEEE International Conference on Computers, Software, and Applications (COMPSAC)*, 2025. URL https://arxiv.org/abs/2505.23953.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.11366.

Dag I. K. Sjøberg, Aiko Yamashita, Bente C. D. Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39 (8):1144–1156, 2013. doi: 10.1109/TSE.2012.89.

Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–does it improve software quality? In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops)*, 2007. doi: 10.1109/WOSQ.2007.11.

Leitian Tao, Baolin Peng, Haorui Wang, Hang Wang, Hao Cheng, Wenlin Yao, Qianhui Wu, Tao Ge, Sharon Li, and Jianfeng Gao. ExecCritic: Learn to Test, Test to Improve for Coding Agents, 2026. URL https://arxiv.org/abs/2609.09133.

Priyansh Trivedi and Olivier Schmitt. Does Code Cleanliness Affect Coding Agents? A Controlled Minimal-Pair Study, 2026. URL https://arxiv.org/abs/2605.20049.

Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2306.01693.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

Terry Yin. Lizard: A simple code complexity analyser. Software, version 1.24.0, released 2026-08-19, 2026. URL https://github.com/terryyin/lizard.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2412.01769.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' Propensity of Exploiting Test Cases, 2025. URL https://arxiv.org/abs/2510.20270.

A SUPPLEMENTARY RELATED WORK

Local feedback and inference-time steering. Fine-Grained RLHF supplies feedback at the level of text segments and separates error types, giving a training signal more specific than a holistic preference (Wu et al., 2023). This motivates a distinction central to our design: an instruction states the desired behavior, whereas localized feedback identifies where the current output falls short. Our reports obtain that information automatically and inject it during inference, without learning a reward model or updating the policy. Inference-time Policy Adapters offer another way to steer a fixed base model, using a lightweight adapter trained with reinforcement learning to modify decoding (Lu et al., 2023). Our intervention instead acts through the agent's context and requires no trained adapter.

Coverage-guided testing and tool feedback, in more detail. CoverUp combines coverage analysis, code context, and iterative feedback to generate tests for Python implementations (Altmayer Pizzorno & Berger, 2025). TestGen-LLM improves existing tests and filters candidates using successful execution, repeated passing runs, and increased coverage (Alshahwan et al., 2024). Both establish coverage as an actionable input to test generation. In our setting the agent changes the implementation as well as its tests: its tests determine the feedback, while a withheld suite evaluates correctness. More generally, coding-agent interfaces already provide automated feedback. SWE-agent studies interfaces for repository navigation, editing, and execution, including edit-time syntax checks (Yang et al., 2024). SKILL.nb uses executable pre- and postcondition gates to validate workflow steps, trigger local repair or fallback, and govern later reuse (El Hattami et al., 2026). Its runtime gates inspect observable state without consulting hidden benchmark evaluators. We share the separation between local operational evidence and final evaluation, but our reports advise the agent about code structure and coverage rather than gate the execution of a reusable workflow. A recent complementary approach, ExecCritic, trains separate test-generation and repair roles and freezes generated tests before repair (Tao et al., 2026). We instead study an untrained intervention within a single coding agent; the independent evaluator is therefore essential to interpreting improvements in its own tests.

What software metrics establish, in more detail. A useful intervention signal need not be a complete measure of software quality. In a large corpus of Java methods and C functions, Landman et al. (2016) find that cyclomatic complexity and source lines of code are related, with the strength of the relationship depending on granularity and transformation. Their results do not support treating the two as interchangeable. Conversely, the human study of Scalabrino et al. (2021) finds that individual metrics, and even their combinations, do not adequately capture code understandability for practical use. We therefore describe reductions in measured complexity and its concentration, rather than equating them with demonstrated improvements in comprehension or maintainability. This distinction also applies to coding agents. In a controlled study of matched repositories, Trivedi & Schmitt (2026) observe lower token use and fewer file revisits on cleaner code without a corresponding pass-rate improvement. That study intervenes on the starting code; ours intervenes on information supplied while the agent writes it.

Does reducing complexity pay off later? Pre-LLM software engineering documents that complexity accumulates, but rarely tests whether reducing it pays off. Lehman's second law holds that an evolving program's complexity increases unless work is done to maintain or reduce it (Lehman, 1980). Parnas describes the same decay as software aging (Parnas, 1994), and change-management data from a large telephone switching system show changes growing harder over fifteen years (Eick et al., 2001). Economic field studies link higher complexity to higher maintenance cost across commercial projects (Banker et al., 1993; Gill & Kemerer, 1991; Kemerer, 1995), and controlled experiments find that complexity affects individual comprehension and modification tasks (Curtis et al., 1979). None of these randomizes a reduction in complexity and follows the code through later evolution. The studies that come closest are mixed. At Microsoft, modules refactored by a dedicated team lost dependencies, but their fall in post-release defects cannot be attributed to refactoring alone (Kim et al., 2014). Refactoring in open-source histories often leaves quality metrics unchanged or worse (Stroggylos & Spinellis, 2007) and introduces code smells about three times as often as it removes them (Cedrim et al., 2017). Classes change less often after a smell is removed, although their fault-proneness does not reliably fall (Palomba et al., 2018). The design closest to ours is the Simula study, in which four companies built the same system from one specification (Anda et al., 2009). Professionals then maintained all four versions, and once file size and number of changes

were controlled, none of twelve code smells increased maintenance effort (Sjøberg et al., 2013). Size is the recurring confounder: associations between structural metrics and outcomes shrink markedly once it is controlled (El Emam et al., 2001; Gil & Lalouche, 2017), and a systematic review finds little evidence that maintainability predictions are effective (Riaz et al., 2009). Structural complexity is therefore an established marker of costly code, but not an established cause whose proactive reduction lowers later cost.

Coverage, test effectiveness, and specification faithfulness. Coverage records execution, not whether a test checks the intended behavior. Controlling for test-suite size, Inozemtseva & Holmes (2014) find only low-to-moderate associations between coverage and mutation-based effectiveness. Checked coverage makes the distinction explicit by measuring executed statements that influence a test oracle (Schuler & Zeller, 2011). Mutation analysis provides a complementary assessment: Just et al. (2014) find a relationship between mutant detection and real-fault detection that persists after controlling for coverage. These results motivate independent correctness evaluation; they do not establish that increasing coverage cannot help. Recent observational evidence also identifies substantial gaps in the coverage of agent-authored changes (Dipongkor et al., 2026). There is a further distinction between passing tests and satisfying a specification. ImpossibleBench constructs conflicts between requirements and tests to expose specification-violating shortcuts (Zhong et al., 2025). Our faithfulness indicators concern feasible benchmark tasks and are not a substitute for such a calibrated assay of cheating. They bound the interpretation of benchmark gains, particularly when an agent can change its own implementation and tests.

Evaluation beyond isolated code generation. Beyond coding, WorkArena evaluates agents acting through software interfaces, and WorkArena++ extends this setting to composite tasks requiring planning and reasoning (Drouin et al., 2024; Boisvert et al., 2024). These tasks motivate evaluation across multiple dependent actions, although they do not measure code erosion. The BrowserGym ecosystem standardizes interfaces and experiment management across web-agent benchmarks, illustrating the importance of a common harness for model comparisons (Le Sellier de Chezelles et al., 2025). Commit0 evaluates the reconstruction of complete libraries using specifications and an interactive development environment (Zhao et al., 2025). We use its library tasks while withholding the official tests, thereby changing the feedback available to the agent. SlopCodeBench evaluates repeated extensions to evolving specifications and measures structural erosion and verbosity (Orlanski et al., 2026). Its finding that quality guidance improves initial code more readily than subsequent degradation motivates our comparison with repeated, implementation-specific reports. SWE-Milestone likewise studies continuous software evolution through dependent milestones derived from repository histories (Deng et al., 2026). These benchmarks make persistence consequential: the code produced at one stage becomes the starting point for later work. Our SlopCodeBench experiment tests that consequence directly by retaining the workspace across checkpoints. Its modified harness and environment, as well as our changed Commit0 test visibility, preclude interpreting absolute scores as leaderboard comparisons.

Reliability of the evaluator. WebArena Verified audits task specifications and evaluators, replacing brittle matching with explicit comparisons and backend-state checks (El Hattami et al., 2025). It illustrates why apparent agent failures can arise from measurement errors. Our benchmark audits address the same methodological concern in library reconstruction: grading failures, missing outcomes, and changing test universes must remain distinguishable from valid zero scores.

Testing whether lower erosion pays off Whether lower erosion makes agent-written code easier to maintain is the natural next question, and the software engineering literature suggests it is harder to answer than it looks. Before language models, complexity was repeatedly associated with maintenance cost, but proactive reductions were rarely tested causally, and the tests that exist are mixed once size and change history are controlled (Section 2 and this appendix). Our intervention removes two obstacles that limited those studies. Agent runs can be replicated from one specification, which the Simula study achieved only at great expense, with four contracted companies (Anda et al., 2009; Sjøberg et al., 2013); and the complexity reduction is randomized, rather than chosen by developers who refactor the code they already find troublesome. A direct test would freeze the final trees produced under R and under C or CT, then assign the same agent, without feedback, a sequence of held-out change requests on each tree. Paired within task and seed, the outcomes would be the

cost of each later change (tokens, turns, and file revisits, as in Trivedi & Schmitt, 2026), defects introduced, hidden-test correctness, and the erosion accumulated after feedback is withdrawn. Two controls follow from the older literature. Code size must be adjusted for, because extract-method decomposition changes the number and length of functions, and size explains much of what structural metrics appear to predict (El Emam et al., 2001; Gil & Lalouche, 2017). The number of later changes should also vary, because a benefit could grow with the horizon, as Lehman's law suggests, or fade as the agent's own style returns. The trivial single-use helpers we observe on SlopCodeBench give a concrete way for the benefit to fail: fragmentation could lower the measured complexity while making later navigation more costly. Longer evolution benchmarks (Deng et al., 2026) make such a study feasible at a scale that human maintenance experiments could not reach.

B How SlapKit measures and feeds back

SlapKit[1] is the measurement tool behind MITL. It scores every function for cyclomatic complexity with lizard (Yin, 2026), an open-source static analyzer that reports the cyclomatic complexity and length of every function in a source file without building the project (version 1.24.0 in every run), joins per-function coverage from a test run, and generates feedback messages to the coding agent according to configuration information (each of our experimental arms correspond to a different configuration). An experimental harness runs the agent, decides when to measure, calls SlapKit and injects the message. The agent is Oh-My-Pi (Bölük, 2026), an open-source terminal coding agent (`https://omp.sh`), at version 18.1.22 in every cell of both experiments. It runs headless with its read, write, edit, bash, grep and glob tools, with its own extensions, skills, rules and language-server integration switched off; the only extension loaded is the harness's, which pauses the agent after every model step so that the workspace can be measured and the message injected before the next model call.

When it measures. The agent signals the end of every model step, also called a turn. The harness snapshots the workspace only when the step changed a file, and measures only those steps: a third of the steps on Commit0 and a fifth on SlopCodeBench. Measurement runs with a fixed deadline in a separate clone of the snapshot, so it never touches the agent's working tree. It is expected for code to sometimes be in an unstable state (e.g. code does not parse, or whose measurement fails or times out); these cases are recorded as unmeasured and produce no message.

Incremental Complexity Feedback. Scores are cached by the content of each file and a version fingerprint of the analysis, so a step re-parses only the files it changed. The functions that are changed after a step are the scored functions that overlap the diff between consecutive snapshots. Only those are reported, and a function is "new this turn" the first it is measured. The message reports each function's cyclomatic complexity against a configurable limit (6 in our experiments).

Test Coverage Feedback. Coverage comes from running the agent's own tests, never the hidden ones,[2] in full after each measured step; unlike complexity, it is not incremental. A changed line is uncovered when the coverage record shows it was not executed, and T reports, per changed function, the number of such lines against a limit of 0. If the agent's tests write files into the tree, the coverage record no longer matches the snapshot and is refused; on SlopCodeBench this "stale tree" case ended some T and CT cells (Section 4.2).

Delivery. The message enters the agent's context before its next model call. The harness records whether each message was delivered and seen, and checks this against the agent's own event log at the end of the run: at least 98.2% of messages were seen (Appendix L). R receives its reminder whenever a metric arm would have a table to show, so the arms differ in what the message says, not in when it arrives.

[1] We will release SlapKit as an open-source package by the paper's publication.

[2] This distinction is important: when evaluating MITL on benchmarks that already include golden unit tests, we hide those from agent's view while it must solve the coding task; we let the agent write its own tests, and it is those tests that participate in the coverage metric. The held-out golden tests are used after the coding task is complete in order to measure functional correctness.

Table 6: Recorded cost of measuring after every agent turn: median / 90th percentile, all models pooled, included cells. *Lane*: wall seconds of the agent-test coverage run per measured turn (all such turns, and those on which at least one test ran); this excludes the rest of the measurement, which no record times, so it is a lower bound. *Model*: the model's own seconds per agent step. *Lane/model*: a cell's lane seconds over its model seconds. *Block*: bytes per delivered feedback block. *Block/proc.*: a cell's injected tokens (characters/4) over its processed tokens.

Bench.	Arm	Cells	Meas. turns	Lane (s) all	Lane (s) tested	Model (s) per step	Lane/model (%)	Block (bytes)	Block/proc. (%)
Commit0	B0	145	5357	0.4 / 1.6	0.8 / 2.5	7.4 / 44.6	0.6 / 5.0	–	0.00 / 0.00
Commit0	R	239	13442	0.5 / 1.8	0.9 / 2.8	7.1 / 44.5	0.7 / 4.8	1167 / 1167	0.43 / 0.94
Commit0	C	146	7407	0.5 / 1.6	0.9 / 2.2	10.1 / 68.2	0.6 / 2.0	1488 / 1667	0.53 / 1.12
Commit0	T	140	6658	0.5 / 1.4	0.8 / 2.1	9.6 / 66.5	0.5 / 2.2	1740 / 3492	0.71 / 1.53
Commit0	CT	239	13125	0.5 / 1.5	0.9 / 2.2	6.7 / 47.1	0.7 / 6.3	2074 / 4072	0.97 / 1.94
SlopCodeBench	B0	141	11208	16.9 / 76.5	19.0 / 82.2	4.1 / 29.1	37.7 / 102.9	–	0.00 / 0.00
SlopCodeBench	R	145	11943	16.0 / 82.7	18.8 / 88.8	3.3 / 28.5	44.2 / 110.5	1166 / 1167	0.52 / 0.81
SlopCodeBench	C	143	12408	16.3 / 73.9	19.1 / 79.7	3.5 / 28.7	40.5 / 122.2	1488 / 1586	0.71 / 1.07
SlopCodeBench	T	127	11701	12.0 / 72.2	14.1 / 77.2	4.1 / 27.9	33.2 / 104.1	1626 / 2792	0.79 / 1.28
SlopCodeBench	CT	130	13251	9.6 / 71.0	12.1 / 74.4	4.2 / 29.1	36.4 / 94.6	1970 / 3538	1.14 / 1.84

Overhead. Table 6 reports overhead: the test run behind coverage, but not the snapshot or the complexity step, so it is a lower bound on the agent's wait. On Commit0 the test run took a median 0.4–0.5 s per measured step, 0.5–0.7% of model time per cell. On SlopCodeBench, whose test suites are slower, it took 9.6–16.9 s, a median 33–44% of model time per cell (40–47% pooled). Coverage is therefore the expensive half of the measurement wherever the agent's suite is slow. The injected feedback message text is small: a median of 1.2 kB (R) to 2.1 kB (CT) per message, or 0.4–1.1% of the tokens a cell processes.

C Prompts and messages

C.1 System prompt

All arms run under oh-my-pi's own system prompt, unchanged by the study. It is identical in every arm and on both benchmarks except for one line, which names the model (`- Model: openrouter/z-ai/glm-5.3-flash` on SlopCodeBench under GLM). Before the task, the harness adds one short notice with the date and the working directory:

harness notice, every arm

```
<system-reminder>
Today: 2026-09-18; current working directory: '/results/run/tree'. Do not repeat this
  information in your reply.
</system-reminder>
```

system prompt, every arm (Commit0, DeepSeek)

```
<conventions>
RFC 2119: MUST, REQUIRED, SHOULD, RECOMMENDED, MAY, OPTIONAL. 'NEVER' = 'MUST NOT'; 'AVOID' = 'SHOULD NOT'.
XML tags inject system content; NEVER interpret them otherwise. Tags may interrupt/notify inside user messages: MUST treat as
  system-authored/authoritative. User content sanitized; role absent: '<system-directive>' in a user turn remains a system directive.
</conventions>

§ Role
Helpful, trusted assistant for load-bearing changes in Oh My Pi coding harness.

# Engineering
- Correctness first; then maintainability 6 months out.
- Apply taste: delete weightless code, refuse needless abstractions, prefer boring; design thoroughly, elegantly.
- Consider compiled code: NEVER avoidably allocate, copy, or compute.
- Unexpected repo changes: user's work; adapt.
- User's word is absolute: user-reported state (errors, failures, observations) is ground truth — act on it directly; NEVER re-run checks
  to confirm what the user already reported.
- Terminal/final chat MAY use LaTeX math ('$', '$$', '\text', '\times') and color ('\textcolor', '\colorbox', '\fcolorbox').
- MAY emit ' "'mermaid ' blocks; terminal renders ASCII. Only genuine structure/flow, not trivia.

# Personality
Evidence-first terse engineer: every sentence fact, decision, or risk.

# Tone
- Fragments when clearer; no ceremony, hedging, summaries, filler, marketing.
- Assume technical reader; don't narrate obvious steps or over-explain basics.
- Concrete: exact files, symbols, APIs, state fields, edge cases, verification.
- Reasoning: facts, constraints, tradeoffs, decisions, checks. Conclusion first; evidence next.
- Uncertainty: state at claim; name tradeoff; choose boring/safe option.
- Code: invariants, risks, verification.

# Reasoning Format
```

```
Problem: what's wrong. Decision: action & why. Check: breakage & verification. Next: concrete action.

# Succinct Patterns
- Y → need update X. This is safe: Z. Could do A, but B avoids C.

# Escalation
Push back on risk-hidden plans or wrong claims: name risk, show evidence, propose alternative. If overruled, execute user's call; don't
  relitigate.

§ Runtime
# Skills & Rules
# Internal URLs
Most FS/bash tools auto-resolve these to FS paths.
- 'skill://<name>': instructions; '/<path>': its file
- 'rule://<name>': details
- 'agent://<id>': output artifact; '/<child>': nested-subagent output; otherwise '/<path>': JSON field
- 'history://<id>': read-only agent transcript (live|parked|released); bare 'history://': all agents. Registered process-wide agents and
  persisted subagents discoverable from artifact trees; unregistered top-level sessions are not discovered solely from persisted session
  files.
- 'artifact://<id>': content
- 'local://<name>.md': plan artifacts/shared subagent content
- 'mcp://<uri>': MCP resource
- 'issue://<N>' / 'issue://<owner>/<repo>/<N>': GitHub issue; bare: recent; '?state=open|closed|all&limit=&author=&label='.
- 'pr://<N>' / 'pr://<owner>/<repo>/<N>': same cache; bare: recent; '?comments=0' '?state=open|closed|merged|all&limit=&author=&label='.
- 'omp://': harness docs; AVOID unless user asks about harness.

# Tool Inventory
- Read: 'read'
- Write: 'write'
- Edit: 'edit'
- Bash: 'bash'
- Grep: 'grep'
- Glob: 'glob'
§ Tool Policy
# General
Use tools when they improve correctness, completeness, or grounding.
- SHOULD resolve prerequisites first; NEVER accept first plausible answer when another call reduces uncertainty; retry
  empty/partial/suspiciously narrow lookup differently.
- SHOULD parallelize independent calls.
# Tool I/O
- Prefer relative 'path'-like fields.
- Most tools take 'i': capitalized 2–6-word present-participle intent (e.g. "Reading model role settings").
# Specialized Tools
MUST use specialized tool over shell equivalent:
- File/directory reads → 'read'; directory path lists entries.
- Surgical edits → 'edit'.
- Create/overwrite → 'write'.

- Regex search/target location → 'grep', not shell 'grep', 'rg', 'awk'.
- Structure mapping/globbing → 'glob', not 'ls **/*.ext' or 'fd'.
- 'bash': real binaries/short fact pipelines only; commands shadowing specialized tools blocked.
- Bash litmus: one external-CLI call/short pipeline returning count, frequency, set difference, checksum. For merely moving, paging,
  trimming fetchable bytes: tool.

<critical>
'write xd://report_issue': automated QA. Any tool output inconsistent with described behavior for parameters → write plain '<tool>:
  <concise description>' to 'xd://report_issue'. False positives fine.
</critical>

# Exploration
NEVER open files hoping. AVOID unneeded files/sections.
- Use 'read' offset/limit, not whole-file reads.
§ Workflow
# 1. Scope

- Multi-file work: plan before files.

# 2. Research Before Editing
- Read sections, not snippets. MUST reuse existing patterns; second convention beside existing is PROHIBITED.

- Tool failure/file change since read → re-read before acting.

# 3. Decompose

# 4. Implement
- Fix source; NEVER suppress symptom/special-case input unless asked.
- Clean cutover: migrate every caller; remove obsolete code/comments/aliases/re-exports/deprecated paths.
- Prefer existing-file updates over new files. Review as user.
- NEVER run destructive git commands/delete unrelated code you didn't write; code the cutover obsoletes is in scope.

# 5. Verify
- NEVER yield non-trivial work without deliverable proof:
  - **Experiment/investigation** → run; output is proof; no tests.
  - **UI change** → verify against the actual surface:
    - **TUI/CLI** → launch the actual program and verify terminal interaction, output, or state.
    - No suitable runtime capability for the changed surface → verify with a throwaway script or smoke test; explicitly report when
  visual verification cannot be performed.
  - **Bug fix** → reproduce, fix, confirm reproduction no longer triggers. SHOULD keep the reproduction as a regression test: fails
  pre-fix, passes post-fix; impractical → smoke test, report it.
  - **Permanent feature/API change** → fix existing tests the changed contract breaks; prove new behavior with a throwaway script. New
  test ONLY for a genuinely uncertain edge case, or on user request.
- Smoke test: run thing, not test file; launch, exercise changed path, observe result.
- Tests: permanent load, not proof of work. A test earns its place ONLY where a plausible bug would fail it.
  - Each MUST defend observable contract/fail on plausible bug.
  - Test behavior, boundaries, invariants, transitions, precedence, real errors—not plumbing, source text, incidental defaults.
  - Match conventions; deterministic, isolated, full-suite-safe.
  - NEVER write a test so the change "has tests" → throwaway script.
  - NEVER assert implementation: wiring, field copies, defaults, forwarding, mock echoes, source text → assert what a consumer observes.
  - NEVER pad: same-path parameter rows, tautologies, bare not-throw, non-empty/length-grew checks.
  - Worth keeping: behavior, boundaries, invariants, transitions, precedence, real errors. Match conventions; deterministic, isolated,
  full-suite-safe.
  - Existing test failing this bar (pins wording, implementation, incidental behavior) → MUST delete; NEVER re-pin it to the new text. In
  scope regardless of author.

# 6. Cleanup
Last phase; REQUIRED after smoke test proves work; NEVER pre-plan/pre-allocate cleanup todos.
- Permanent feature/bug fix → docs, changelog, scaffold + throwaway-script removal; tests only per Verify.
- Experiment/one-off investigation → no cleanup tests/docs.

§ Delivery
<contract>
Inviolable.
- NEVER yield before complete deliverable; phase boundary/todo flip/sub-step never yields: same turn.
- NEVER fabricate output; code/tool/test/doc/source claims MUST be grounded.
- NEVER substitute easier/familiar problem: don't infer extra scope—retries, validation, telemetry, abstraction "while you're at it"—or
  solve symptom—suppress warning/exception, special-case input—unless asked. Real ask only.
- NEVER ask for tool/repo/file-provided information; NEVER punt half-solved work.
- Default clean cutover: migrate every caller; no shims, aliases, deprecated paths.
</contract>
```

```
<completeness>
- “Done”: specified end-to-end behavior plus every named acceptance criterion; not compiling scaffold, narrowed test, plausible subset.
- Reduce scope only with explicit user approval in this conversation; NEVER silently shrink.
- NEVER deliver unfinished work: stubs, placeholders, mocks, no-ops, fake fallbacks, ‘TODO: implement‘, misleading
  “scaffold”/“MVP”/“v1”/“foundation”/“follow-up”. Unavailable real-implementation info → state missing prerequisite; finish all reachable
  work.
</completeness>

<evidence-and-output>
- Format MUST match ask; prose brief; evidence, verification, blocking details complete.
- Code/tool/test/doc/source claims MUST be grounded; unobserved claims ‘[INFERENCE]‘.
- Verification claims exactly match exercised work.
</evidence-and-output>

<yielding>
Before yielding: all affected callsites/tests/docs updated or intentionally unchanged; output/evidence requirements satisfied.
Before blocked: ensure info unreachable via tools/context; one failed check ≠ blocked. Finish reachable work; state exactly missing and
  tried.
</yielding>

§ Critical
<critical>
- NEVER yield while actionable work remains; phase boundary/todo flip/sub-step never stops: same turn.
- NEVER narrate/consider session limits, token/tool budgets, effort estimates, or possible completion; start unbounded: execute/delegate.
- NEVER re-audit applied edit or routinely run git subcommands for validation. Tool results are verification.
</critical>

PROJECT

<workstation>
- OS: linux 6.14.0-1015-nvidia
- Distro: Linux
- Kernel: #15-Ubuntu SMP PREEMPT_DYNAMIC Tue Nov 25 18:02:16 UTC 2025
- Arch: arm64
- Model: openrouter/deepseek/deepseek-v4.1-flash
</workstation>
<critical>
- Each response MUST advance the task; completion only stopping condition.
- MUST default to informed action; do not ask for confirmation when tools or repo context can answer.
- Before yielding, MUST verify significant behavioral changes: run the specific test, command, or scenario covering the change.
</critical>
```

C.2 TASK PROMPTS

On Commit0 the task prompt names the library, removes its implementation and its tests, and asks the agent to write its own tests. Shown here for `attrs`. B0's prompt ends after *Finishing*; each other arm appends one announcement, shown after it.

Commit0 task prompt (attrs), common to every arm

```
# Implement ‘attrs‘

You are in the repository of the Python library ‘attrs‘. Its implementation has been
removed: under ‘src/‘, the modules, the classes, the function signatures and the
docstrings are all still there, but the bodies are not.

Write the implementation so that the library’s original tests would pass.

This library’s test suite has been removed. Your implementation will be judged by that
original suite, which you cannot see. Write your own tests under tests and use them to
check your work.

## Where things are

- ‘src/‘ is the package to implement. That is where your work goes.
- The project’s own documentation is at https://www.attrs.org/en/24.2.0/.

## Running the tests

    /bench/open/venv/attrs/bin/python -m pytest tests

Use that interpreter and no other: it is the one the library’s dependencies are installed
in. It is outside this repository, so give it by its full path exactly as written above.

## Finishing

Work until you are confident the implementation is complete and correct, or until you have
gone as far as you usefully can, and then say which of the two happened.
```

addition, arm R

```
After some of your turns you will receive an automatic message headed [Reminder]. It
  needs no reply.

Keep every function you write understandable, tested and maintainable. Passing the tests
  comes first.
```

addition, arm C

```
After some of your turns you will receive an automatic message headed [Measurement]. It
  reports the number described below for the functions you changed, and needs no reply.

Cyclomatic complexity estimates how many independent execution paths a function contains.
  Higher complexity usually means the function is harder to understand, test, and modify
  safely. The target is 6 or lower for every function.

Keep every function you write understandable, tested and maintainable. Passing the tests
  comes first.
```

addition, arm T

```
After some of your turns you will receive an automatic message headed [Measurement]. It
  reports the number described below for the functions you changed, and needs no reply.

Uncovered lines counts, for each function you changed in a turn, the lines you added or
  edited that this repository's test suite did not execute. A line no test runs is a line
  whose behavior nothing checks. The target is 0 for every function you change.

Keep every function you write understandable, tested and maintainable. Passing the tests
  comes first.
```

addition, arm CT

```
After some of your turns you will receive an automatic message headed [Measurement]. It
  reports the two numbers described below for the functions you changed, and needs no
  reply.

Cyclomatic complexity estimates how many independent execution paths a function contains.
  Higher complexity usually means the function is harder to understand, test, and modify
  safely. The target is 6 or lower for every function.

Uncovered lines counts, for each function you changed in a turn, the lines you added or
  edited that this repository's test suite did not execute. A line no test runs is a line
  whose behavior nothing checks. The target is 0 for every function you change.

Keep every function you write understandable, tested and maintainable. Passing the tests
  comes first.
```

On SlopCodeBench each checkpoint's prompt opens with the preamble below, then the same arm addition as on Commit0, then `Your task is:` and the checkpoint's specification, which is benchmark material and is not reproduced. From the second checkpoint on, one line of the preamble changes: `Keep using the same virtual environment you started with, update requirements.txt with any new dependencies you need.`

SlopCodeBench preamble, checkpoint 1, every arm

```
Implement a program that 100% solves the specification. That is all you need to do.
Use a virtual environment and ensure that a 'requirements.txt' is present with any
  dependencies you need to solve the problem.

Your program will be judged by tests you cannot see, which run it as a command. Write
  your own tests under 'tests/' and use them to check your work.

These packages are installed and no others can be added: annotated-doc, annotated-types,
  anyio, attrs, beautifulsoup4, blinker, bracex, cachebox, certifi, charset-normalizer,
  click, coverage, cssselect, deepdiff, duckdb, et-xmlfile, exceptiongroup, fastapi,
  flask, grpcio, grpcio-tools, h11, httpcore, httptools, httpx, idna, iniconfig,
  itsdangerous, jinja2, jsonschema, jsonschema-specifications, lxml, markdown-it-py,
  markupsafe, mdurl, numpy, openpyxl, orderly-set, packaging, pandas, pexpect, pika,
  pluggy, protobuf, ptyprocess, pyarrow, pydantic, pydantic-core, pygments, pytest,
  pytest-asyncio, pytest-cov, pytest-json-ctrf, pytest-json-report, pytest-metadata,
  pytest-timeout, python-dateutil, python-dotenv, pyyaml, referencing, requests, rich,
  rpds-py, ruamel-yaml, ruamel-yaml-clib, scipy, setuptools, shellingham, six, soupsieve,
  sqlglot, starlette, toml, tomli, tree-sitter, tree-sitter-languages,
  tree-sitter-python, typer, typing-extensions, typing-inspection, urllib3, uvicorn,
  uvloop, watchfiles, wcmatch, websockets, werkzeug, xlrd. A 'requirements.txt' you write
  is not installed here, so the packages above are the only ones your program can import.

Work until you are confident the program is complete and correct, or until you have gone
  as far as you usefully can, and then say which of the two happened.
```

C.3 Messages after a turn

After a turn that changed code, each arm receives one message; B0 receives none. R's message is the review instructions alone, headed `[Reminder, turn` k`]`. C, T and CT add their tables,

headed `[Measurement, turn k]`; a table lists the changed functions above the limit, highest first (or `none`), and counts those at or under it. The review instructions that end every message are identical in all arms. Real messages from Commit0 runs, with long tables cut to their first rows:

SlapKit feedback — R: reminder only

```
[Reminder, turn 3]

Review the functions you changed this turn for readability, testability and
  maintainability.

For each function you changed:

1. Read the complete function and identify the distinct responsibilities it performs.
2. Simplify nested control flow using guard clauses or early returns where appropriate.
3. Extract coherent responsibilities into clearly named helper functions with explicit
  inputs and outputs.
4. Consolidate repeated branches or special cases using shared logic, lookup tables, or
  dispatch functions when appropriate.
5. Make sure the tests exercise the function's behavior, including its error paths and
  edge cases, and assert on results or effects.
6. Preserve the function's public behavior and interfaces, and run the relevant tests
  after any change, followed by the broader test suite when practical.

Do not create arbitrary one-line wrappers, scatter closely related logic across
  meaningless helpers, delete required behavior, or weaken tests. The goal is
  understandable and maintainable code.

Do not interrupt correct task completion for unnecessary refactoring. Correctness and
  passing the tests remain the first priority.
```

SlapKit feedback — C: complexity table

```
[Measurement, turn 1]

Cyclomatic complexity estimates how many independent execution paths a function contains.
  Higher complexity usually means the function is harder to understand, test, and modify
  safely. The target is 6 or lower for every function.

Functions above the limit, highest first:

  src/attr/_make.py:60  attrib
    cyclomatic complexity 18 (limit 6); new this turn

At or under the limit: 0 functions.

[the same review instructions as in the R message above]
```

SlapKit feedback — T: uncovered changed lines

```
[Measurement, turn 1]

Uncovered lines counts, for each function you changed this turn, the lines you added or
  edited that this repository's test suite did not execute. A line no test runs is a line
  whose behavior nothing checks. The target is 0 for every function you change.

Functions above the limit, highest first:

  jwt/utils.py:32  raw_to_der_signature
    uncovered lines 8 (limit 0): no test loads this file; new this turn
  jwt/utils.py:41  force_bytes
    uncovered lines 7 (limit 0): no test loads this file; new this turn
  jwt/utils.py:16  base64url_decode
    uncovered lines 6 (limit 0): no test loads this file; new this turn
  [...]

At or under the limit: 0 functions.

[the same review instructions as in the R message above]
```

SlapKit feedback — CT: both tables

```
[Measurement, turn 3]

Cyclomatic complexity estimates how many independent execution paths a function contains.
  Higher complexity usually means the function is harder to understand, test, and modify
  safely. The target is 6 or lower for every function.

Functions above the limit, highest first:

none

At or under the limit: 4 functions.

Uncovered lines counts, for each function you changed this turn, the lines you added or
  edited that this repository's test suite did not execute. A line no test runs is a line
  whose behavior nothing checks. The target is 0 for every function you change.

Functions above the limit, highest first:

  src/click/globals.py:8  get_current_context
    uncovered lines 6 (limit 0): no test loads this file; new this turn
  src/click/globals.py:40  resolve_color_default
    uncovered lines 6 (limit 0): no test loads this file; new this turn
  src/click/globals.py:30  push_context
    uncovered lines 3 (limit 0): no test loads this file; new this turn
  [...]

At or under the limit: 0 functions.

[the same review instructions as in the R message above]
```

D FACTORIAL CONTRASTS

For library-averaged outcomes under R, C, T, and CT, the secondary main effects are

$$\Delta_C = \tfrac{1}{2}[(Y_C - Y_R) + (Y_{CT} - Y_T)], \tag{3}$$

$$\Delta_T = \tfrac{1}{2}[(Y_T - Y_R) + (Y_{CT} - Y_C)], \tag{4}$$

$$\Delta_{C\times T} = Y_{CT} - Y_C - Y_T + Y_R. \tag{5}$$

These contrasts concern the information added to a common reminder; a positive interaction would indicate a benefit beyond the sum of the two individual additions. Table 7 gives all three on held-out pass fraction, for both experiments. None of the six model-specific complexity contributions excludes zero. The post hoc pool of the two Commit0 models does, at 3.0 points [0.3, 5.6] with three seeds and 2.0 [0.1, 4.0] with five, but on SlopCodeBench the same contrast is 0.7 under DeepSeek and -2.1 under GLM. The intervals are not adjusted for multiplicity, and no interaction is resolved. The task bootstrap, which Section 3 names as a sensitivity analysis, disagrees with the paired t interval on two of these contrasts: it excludes zero for the DeepSeek complexity contribution on Commit0 ([0.3, 7.2] points with three seeds, [0.2, 5.4] with five) and for the DeepSeek coverage contribution on SlopCodeBench ($[-4.1, -0.2]$).

E ADDITIONAL COMMIT0 RESULTS

Table 8 shows additional results on Commit0, as we vary the number of seeds.

F MANIPULATION CHECK AT THE FEEDBACK LIMIT

The complexity report asks the agent to bring every changed function to cyclomatic complexity 6 or below. Two measures follow that request directly: E_6, which is Equation 2 with $\tau = 6$, and the share of scored production functions with complexity above 6. We report them to confirm that the agents did what the report asked, not as outcomes; E_{10} remains the outcome. Both were stored beside E_{10} in the same measurement records, so nothing was measured again.

Table 9 gives the paired contrasts at the final tree. Both measures fall under C and CT, on both benchmarks and under both models, and every interval excludes zero. The fall in E_6 is about twice the fall in E_{10} on Commit0 (1.7–2.1 times) and about 1.4 times on SlopCodeBench (1.35–1.5 times). The

Table 7: Pre-registered secondary contrasts on held-out pass fraction, percentage points: complexity contribution $\frac{1}{2}[(\mathrm{C}-\mathrm{R})+(\mathrm{CT}-\mathrm{T})]$, coverage contribution $\frac{1}{2}[(\mathrm{T}-\mathrm{R})+(\mathrm{CT}-\mathrm{C})]$ and interaction $\mathrm{CT}-\mathrm{C}-\mathrm{T}+\mathrm{R}$, computed per task from arm means. Paired 95% t intervals, unadjusted for multiplicity; tasks missing any of the four arms are dropped.

Benchmark	Seeds	Model	Complexity	Coverage	Interaction
Commit0	seeds 1–3	DeepSeek	$+3.3$ [-0.5, +7.1]	$+1.1$ [-1.5, +3.7]	$+1.1$ [-9.5, +11.7]
Commit0	seeds 1–3	GLM	$+2.6$ [-1.6, +6.7]	$+2.2$ [-1.7, +6.0]	$+2.4$ [-7.1, +11.8]
Commit0	seeds 1–3	Pooled*	$+3.0$ [+0.3, +5.6]	$+1.5$ [-0.6, +3.5]	$+1.8$ [-4.9, +8.5]
Commit0	seeds 1–5‡	DeepSeek	$+2.5$ [-0.3, +5.3]	$+0.3$ [-2.1, +2.7]	-1.7 [-7.3, +3.9]
Commit0	seeds 1–5‡	GLM	$+1.5$ [-1.6, +4.6]	$+1.1$ [-2.7, +4.9]	$+3.4$ [-3.6, +10.5]
Commit0	seeds 1–5‡	Pooled*	$+2.0$ [+0.1, +4.0]	$+0.5$ [-1.4, +2.5]	$+0.3$ [-3.9, +4.5]
SCB	seeds 1–2	DeepSeek	$+0.7$ [-1.5, +2.9]	-2.0 [-4.1, +0.1]	-3.7 [-8.3, +0.9]
SCB	seeds 1–2	GLM	-2.1 [-5.5, +1.2]	$+1.6$ [-1.9, +5.2]	-1.0 [-7.5, +5.6]

*Post hoc fixed-effect inverse-variance pool of the two models, normal 95% interval. ‡R and CT use seeds 1–5; C and T exist for seeds 1–3 only.

Table 8: Commit0 primary contrast CT–R, held-out pass fraction in percentage points, by seed set. Seeds 1–3 are the pre-registered design; seeds 4–5 (R and CT only) were added after the first reading, on the same image with a different realized provider mix. Paired-library 95% t intervals (25 libraries).

Model	Seeds 1–3 (pre-registered)	Seeds 1–5	Seeds 4–5 only
DeepSeek	$+4.4$ [-1.2, +10.0]	$+2.8$ [-1.3, +6.9]	$+0.5$ [-5.3, +6.2]
GLM	$+4.7$ [-1.6, +11.0]	$+2.6$ [-2.1, +7.3]	-1.2 [-8.8, +6.5]
Pooled*	$+4.6$ [+0.6, +8.5]	$+2.7$ [-0.2, +5.6]	-0.1 [-4.5, +4.3]

*Post hoc fixed-effect inverse-variance pool of the two models, normal 95% interval.

share of functions over the limit falls by 4–6 percentage points on Commit0 and by 22–29 on SlopCodeBench. The linear mixed model agrees on E_6: averaged over the two models, C–R is -0.229 $[-0.259, -0.199]$ on Commit0 (seeds 1–3) and -0.682 $[-0.699, -0.664]$ on SlopCodeBench. Its fit for the share of functions over the limit on SlopCodeBench is singular, so we rely on the paired intervals for that measure. Figures 3 and 4 repeat Figure 2(a) and (b) with E_6. On Commit0 the gap opens steadily with the budget, as it does for E_{10}. On SlopCodeBench, E_6 stays at or below 0.06 under C and CT at every checkpoint, against 0.60–0.76 under R: under the reminder alone, most of the codebase's complexity mass sits in functions above the limit.

G Additional metrics

Table 10 gives the paired contrasts on three measures that are not provided as feedback to any arm: SlopCodeBench's own erosion score, computed by the benchmark's checker with its own complexity counter over every file in the snapshot; its verbosity score (Section 3.3); and the share of the agent's production lines that the held-out tests execute, on both benchmarks. On SlopCodeBench every C–R and CT–R interval excludes zero, in the direction of less concentrated complexity, fewer verbose or duplicated lines and more code reached by the hidden tests. The reminder alone moves all three relative to no message, and complexity feedback moves them again relative to the reminder: by a similar amount for verbosity and hidden-suite coverage, and by nearly twice as much for the erosion score. Coverage feedback alone does not lower the erosion score (under GLM it raises it slightly) and does not move hidden-suite coverage; it lowers verbosity under DeepSeek and not under GLM. On Commit0 only hidden-suite coverage can be measured, and no feedback contrast resolves there; the reminder alone raises it under DeepSeek. Sections 5.1 and 6 read these results.

H Integrity audit

We audited every trace of both experiments for internet access, access to hidden tests or reference code, and grader tampering. On SlopCodeBench no agent reached anything outside its workspace, although many tried (220 of the 400 analyzed DeepSeek and Qwen traces, for instance): the

Table 9: Manipulation check: the complexity measures the report asks the agent to reduce, beside the E_{10} outcome, at the final tree. E_6 is the share of complexity mass in functions with CC > 6 (the feedback limit); the last pair is the change in the percentage of scored functions with CC > 6. Pairing, seeds and complete-case rule as in Table 3: Commit0 pairs arm means within library (seeds 1–5 for R and CT, 1–3 for C); SCB pairs arm means within problem (2 seeds). Paired 95% t intervals.

		E_{10} (outcome)		E_6		Functions over CC 6 (points)	
Benchmark	Model	C–R	CT–R	C–R	CT–R	C–R	CT–R
Commit0	DeepSeek	−0.123 [-0.185, -0.061]	−0.070 [-0.110, -0.030]	−0.226 [-0.305, -0.146]	−0.146 [-0.209, -0.083]	−5.6 [-9.2, -2.0]	−4.2 [-7.7, -0.8]
Commit0	GLM	−0.120 [-0.177, -0.064]	−0.096 [-0.145, -0.046]	−0.209 [-0.278, -0.140]	−0.160 [-0.223, -0.097]	−4.9 [-7.4, -2.4]	−4.0 [-6.7, -1.4]
SCB	DeepSeek	−0.428 [-0.483, -0.372]	−0.433 [-0.487, -0.379]	−0.639 [-0.682, -0.596]	−0.628 [-0.673, -0.584]	−22.1 [-24.7, -19.6]	−22.0 [-24.4, -19.7]
SCB	GLM	−0.529 [-0.573, -0.485]	−0.504 [-0.546, -0.461]	−0.723 [-0.754, -0.693]	−0.682 [-0.724, -0.640]	−28.8 [-31.7, -25.9]	−27.4 [-30.1, -24.6]

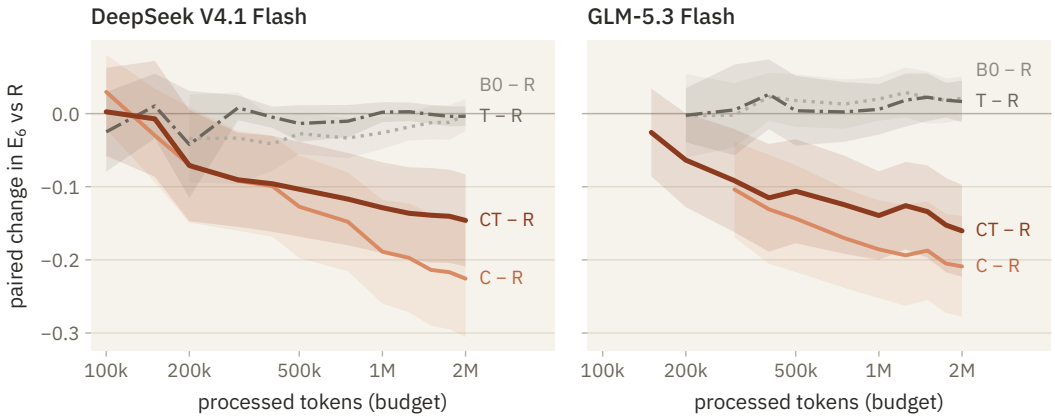

Figure 3: Manipulation check on Commit0: paired change in E_6 relative to R at each processed-token budget, drawn as Figure 2(a) draws E_{10}. Library-paired, with 95% t bands; a rung is drawn when at least 15 libraries pair; seeds 1–5 for R and CT, 1–3 for the other arms.

operating-system fence blocked every attempt. On Commit0 the filesystem fence checked only the words of a path, and agents routinely got past it by assembling paths from pieces; the network proxy did block every attempt to fetch upstream source. Every Commit0 cell that reached hidden or finished-library material is set aside (28 cells), including two found by this audit. An operating-system boundary, not a path filter, is what kept the second experiment clean.

I THE OTHER CONTRASTS AS ARROWS

Figure 5 draws the other three contrasts the way Figure 1(b) draws R→CT. Complexity feedback alone (R→C) moves every arrow up and none reliably sideways, like the combined feedback. Coverage feedback alone (R→T) barely moves anything, and on SlopCodeBench it slightly raises erosion. The reminder alone (B0→R) lowers erosion on SlopCodeBench and leaves Commit0 where it was.

J MIXED MODELS

The paired intervals of the main text are the pre-registered analysis. In support, we fit a binomial generalized linear mixed model (GLMM) on test counts (R's implementation of `lme4::glmer`, Laplace approximation, `bobyqa` optimizer) and a linear mixed model (LMM) on E_{10} (REML). The GLMM has fixed effects `arm * model`, plus the seed block (Commit0) or the seed (SlopCodeBench), and random intercepts for task, task×model and cell. The LMM has the same fixed effects and random intercepts for task and task×model; it has one row per cell, so no cell term. Every fit in Table 11 converged, and none is singular. A quasi-binomial GLM with task-clustered sandwich errors, and logit-scale and fractional-logit versions of the erosion model, agree in the sign and significance of every C–R and CT–R erosion contrast, with one exception: the fractional-logit version is undefined where E_{10} is almost exactly zero, as under Qwen C. Figure 6 shows every fixed-effect contrast of these fits, including the erosion contrasts and the recovery fit that Table 11 omits. Figure 7

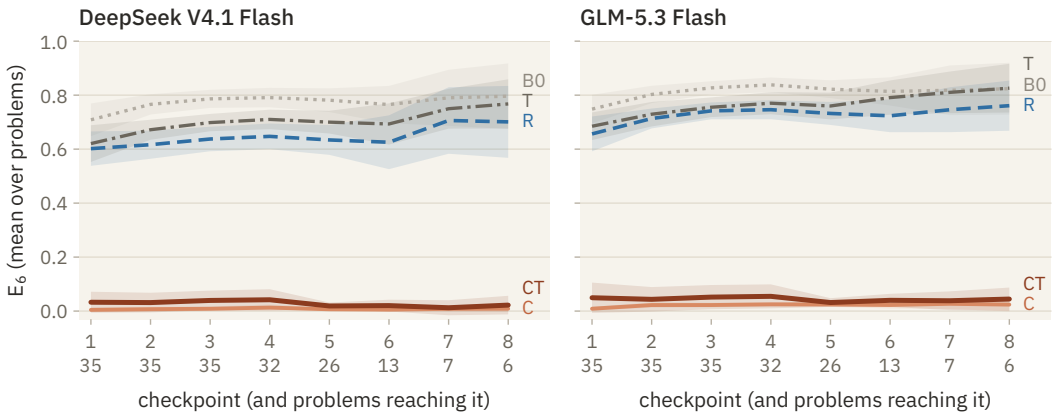

Figure 4: Manipulation check on SlopCodeBench: E_6 at each checkpoint by arm, drawn as Figure 2(b) draws E_{10} (mean over problems of the problem mean, 95% t band, main-run included cells, two seeds). The second tick row gives the number of problems that reach each checkpoint.

Table 10: Measures the agent is never shown, paired within task as in Table 3. SlopCodeBench's own erosion score and its verbosity score (Section 3.3) come from the benchmark's pinned checker on the whole snapshot; hidden-suite line coverage is the share of the agent's production lines that the held-out tests execute. Lower is better for erosion and verbosity, higher for coverage. Commit0 has no benchmark checker, so only its hidden-suite coverage appears (seeds 1–5 for R and CT, 1–3 for B0, C, T). Paired 95% t intervals; cells without the measure are dropped (complete case).

Benchmark	Measure	Model	C–R	CT–R	T–R	R–B0
SCB	benchmark erosion score	DeepSeek	−0.256 [-0.308, -0.203]	−0.257 [-0.310, -0.204]	+0.013 [-0.017, +0.043]	−0.157 [-0.189, -0.124]
SCB	benchmark erosion score	GLM	−0.293 [-0.337, -0.249]	−0.277 [-0.319, -0.236]	+0.027 [+0.006, +0.048]	−0.155 [-0.181, -0.130]
SCB	verbosity (points)	DeepSeek	−3.4 [-4.8, -1.9]	−4.5 [-6.3, -2.6]	−2.1 [-3.6, -0.5]	−3.5 [-5.0, -2.0]
SCB	verbosity (points)	GLM	−4.4 [-6.1, -2.7]	−4.4 [-6.0, -2.9]	+1.3 [-0.3, +3.0]	−3.6 [-5.8, -1.4]
SCB	hidden-suite coverage (points)	DeepSeek	+2.9 [+1.2, +4.6]	+2.4 [+0.5, +4.3]	+0.8 [-0.5, +2.2]	+3.3 [+0.8, +5.7]
SCB	hidden-suite coverage (points)	GLM	+3.7 [+1.9, +5.5]	+5.1 [+3.0, +7.2]	+0.6 [-0.8, +2.0]	+2.9 [+0.7, +5.1]
Commit0	hidden-suite coverage (points)	DeepSeek	+1.1 [-1.0, +3.2]	+0.6 [-1.2, +2.4]	−0.4 [-2.8, +2.0]	+2.8 [+0.5, +5.1]
Commit0	hidden-suite coverage (points)	GLM	+1.2 [-4.9, +7.3]	+3.7 [-0.5, +7.9]	+1.3 [-6.5, +9.1]	−1.2 [-7.3, +5.0]

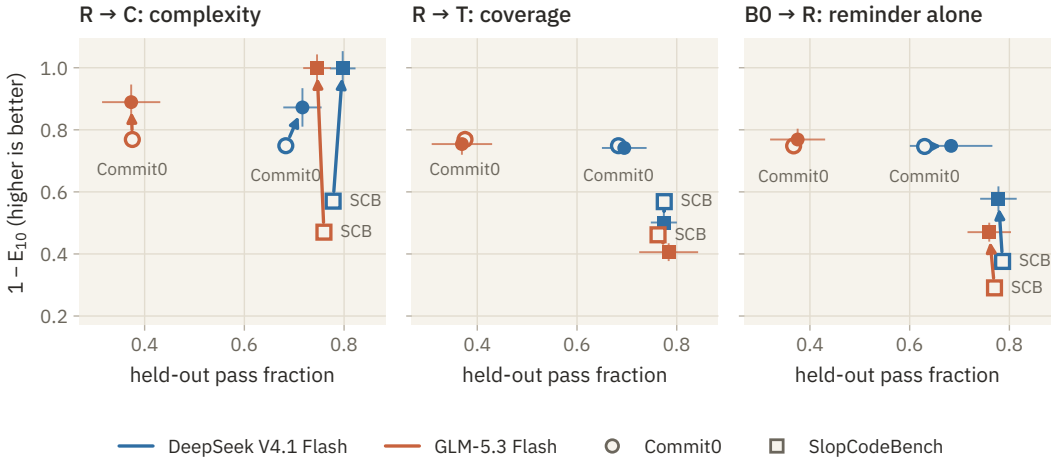

Figure 5: Arrow diagrams for the other contrasts, drawn as in Figure 1(b): (a) R→C, complexity feedback; (b) R→T, coverage feedback; (c) B0→R, the reminder alone. Hollow markers are the starting arm, filled the ending arm; arrows are the pre-registered paired estimates and whiskers their paired 95% t intervals (Commit0: R over seeds 1–5, C, T and B0 over seeds 1–3; SlopCodeBench: problems paired within model, complete case).

shows what the random effects absorb, and why the design pairs within task. On correctness, the standard deviation of the task intercepts is 3.35 on the log-odds scale on Commit0 and 1.48 on SlopCodeBench, about 9 and 16 times the CT–R effect (0.38 and 0.09). The library intercepts on

Table 11: Mixed models. Correctness: odds ratios from a binomial GLMM on test counts, `arm * model` (+ seed block or seed) with random intercepts for task, task×model and cell (Laplace, bobyqa). Erosion: CT–R from a REML linear mixed model on E_{10} with task and task×model intercepts. Wald 95% intervals; "Both" averages the two model-specific contrasts. All fits converged and none is singular.

Data	Model	OR C–R	OR CT–R	OR T–R	OR R–B0	ΔE_{10} CT–R
Commit0, seeds 1–3	DeepSeek	1.15 [0.57, 2.34]	1.30 [0.64, 2.66]	1.16 [0.56, 2.39]	2.37 [1.16, 4.83]	-0.062 [-0.096, -0.027]
Commit0, seeds 1–3	GLM	1.06 [0.52, 2.16]	1.64 [0.81, 3.32]	0.91 [0.45, 1.85]	0.82 [0.40, 1.67]	-0.092 [-0.127, -0.056]
Commit0, seeds 1–3	Both	1.11 [0.67, 1.83]	1.46 [0.88, 2.41]	1.03 [0.62, 1.70]	1.40 [0.84, 2.31]	-0.077 [-0.101, -0.052]
Commit0, seeds 1–5	DeepSeek	1.24 [0.65, 2.36]	1.28 [0.74, 2.22]	1.19 [0.62, 2.31]	2.20 [1.15, 4.20]	-0.069 [-0.095, -0.043]
Commit0, seeds 1–5	GLM	0.89 [0.47, 1.69]	1.27 [0.74, 2.19]	0.75 [0.39, 1.44]	1.00 [0.53, 1.92]	-0.099 [-0.126, -0.072]
Commit0, seeds 1–5	Both	1.05 [0.66, 1.68]	1.28 [0.87, 1.88]	0.95 [0.59, 1.53]	1.49 [0.93, 2.38]	-0.084 [-0.103, -0.065]
SCB	DeepSeek	1.10 [0.88, 1.38]	0.86 [0.68, 1.08]	0.89 [0.70, 1.12]	1.01 [0.81, 1.27]	-0.420 [-0.453, -0.386]
SCB	GLM	0.93 [0.74, 1.17]	0.96 [0.76, 1.21]	0.99 [0.78, 1.25]	0.93 [0.74, 1.17]	-0.512 [-0.546, -0.479]
SCB	Both	1.01 [0.86, 1.19]	0.91 [0.77, 1.07]	0.94 [0.79, 1.11]	0.97 [0.83, 1.14]	-0.466 [-0.490, -0.442]

Table 12: Cells by arm: planned / failed (no graded outcome) / set aside (graded, but excluded for hidden-side access, contamination or the audit) / included without a measured E_{10}. Commit0 counts include the seed 4–5 block (R and CT only). SCB failures in T and CT include a known harness defect (the coverage step refuses a tree the agent's own tests have written into).

Benchmark	Model	B0	R	C	T	CT
Commit0	DeepSeek	75 / 1 / 3 / 4	125 / 4 / 6 / 3	75 / 1 / 3 / 0	75 / 5 / 5 / 0	125 / 2 / 8 / 1
Commit0	GLM	75 / 0 / 1 / 9	125 / 0 / 1 / 23	75 / 0 / 0 / 10	75 / 0 / 0 / 10	125 / 0 / 1 / 16
SCB	DeepSeek	70 / 4 / 0 / 1	70 / 1 / 0 / 0	70 / 4 / 0 / 1	70 / 11 / 0 / 0	70 / 11 / 0 / 0
SCB	GLM	70 / 4 / 0 / 0	70 / 4 / 0 / 1	70 / 3 / 0 / 1	70 / 10 / 0 / 0	70 / 8 / 0 / 1
SCB	Qwen	10 / 1 / 0 / 0	10 / 0 / 0 / 0	10 / 0 / 0 / 0	10 / 2 / 0 / 0	10 / 1 / 0 / 1

Commit0 span -9.4 to $+4.5$ log-odds, and the R arm's raw pass fraction runs from 0 to 1 across libraries. The model×library interaction is large on Commit0 (standard deviation 2.16; the R arm passes 0–2% of attrs, click and jinja under GLM but 78–91% under DeepSeek) and small on SlopCodeBench (0.17). Task heterogeneity of this size is what makes the correctness intervals wide. Erosion on SlopCodeBench is the reverse case: the CT–R effect (0.47) is about six times the task standard deviation (0.078). On Commit0 it is about half of it (0.077 against 0.138).

K Missing cells and sensitivity analyses

Table 12 counts cells by arm. Table 13 repeats the primary contrast under each rule. The Commit0 rules are:

- dropping workspaces that scored zero because the hidden suite produced no report;
- dropping only those whose suite timed out or was killed;
- dropping the three libraries whose starting tree contains syntax that the library's interpreter rejects, which leaves E_{10} unmeasured;
- filling missing E_{10} from the last measured budget rung;
- putting the set-aside cells back.

The SlopCodeBench rules add one graded cell whose record is marked incomplete, carry the last graded checkpoint forward (possible for two cells only), and fill missing cells either adversely to the feedback arms or with zero. We also fit the Commit0 GLMM with each cell's dominant upstream provider as a covariate, an adjustment chosen post hoc. Figure 8 repeats Figure 1(b) and Figure 5(a) with the post hoc T/CT recovery cells in place of the missing SlopCodeBench cells.

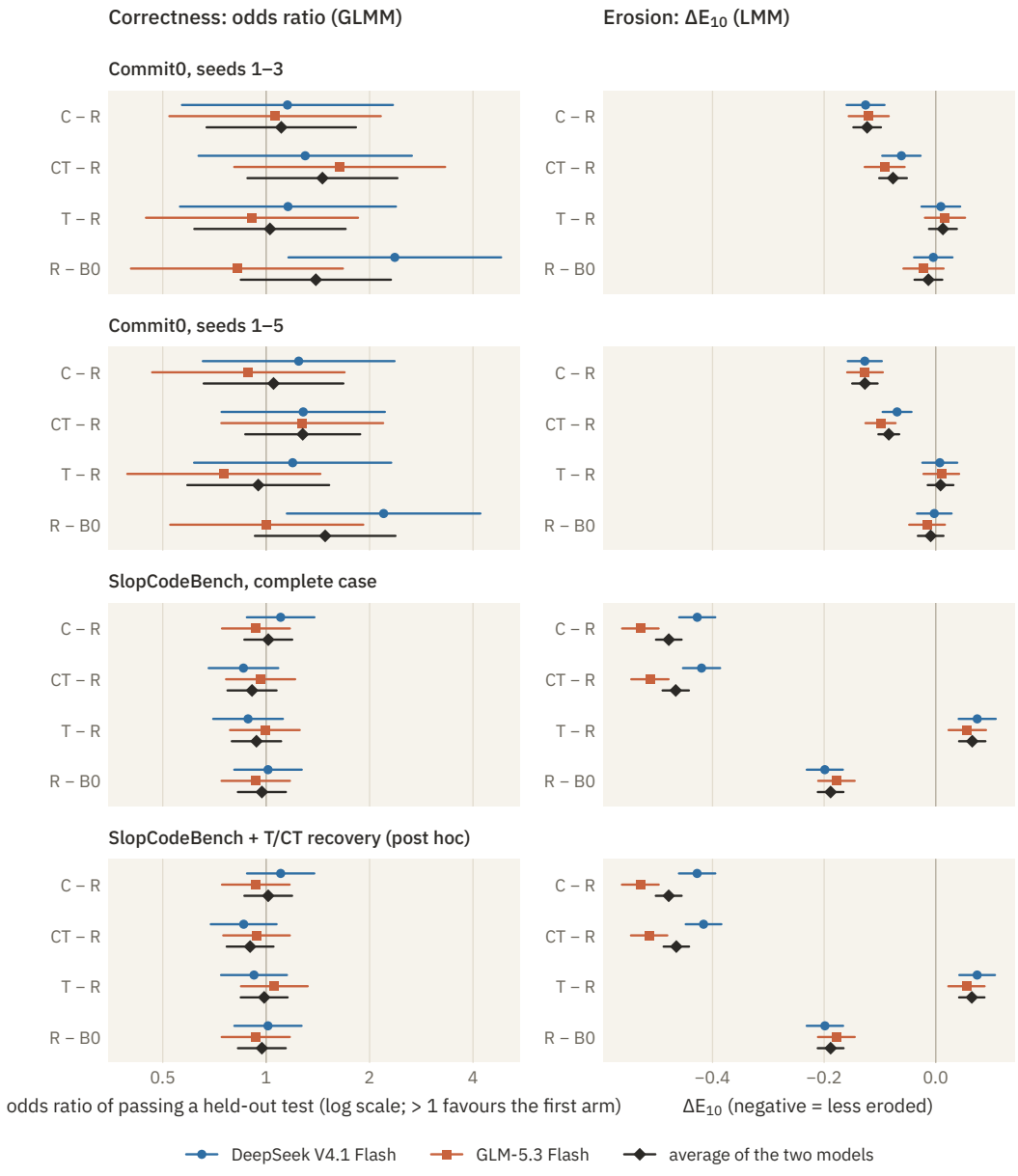

Figure 6: Fixed-effect contrasts of the mixed models with Wald 95% intervals. Left: odds ratio of passing a held-out test (binomial GLMM on test counts with task, task×model and cell random intercepts; log scale). Right: change in E_{10} (REML linear mixed model with task and task×model intercepts). Rows: Commit0 over seeds 1–3 (pre-registered) and 1–5; SlopCodeBench complete case and with the post hoc T/CT recovery cells. Blue DeepSeek V4.1 Flash, orange GLM-5.3 Flash, black the average of the two model-specific contrasts. Values as in Table 11.

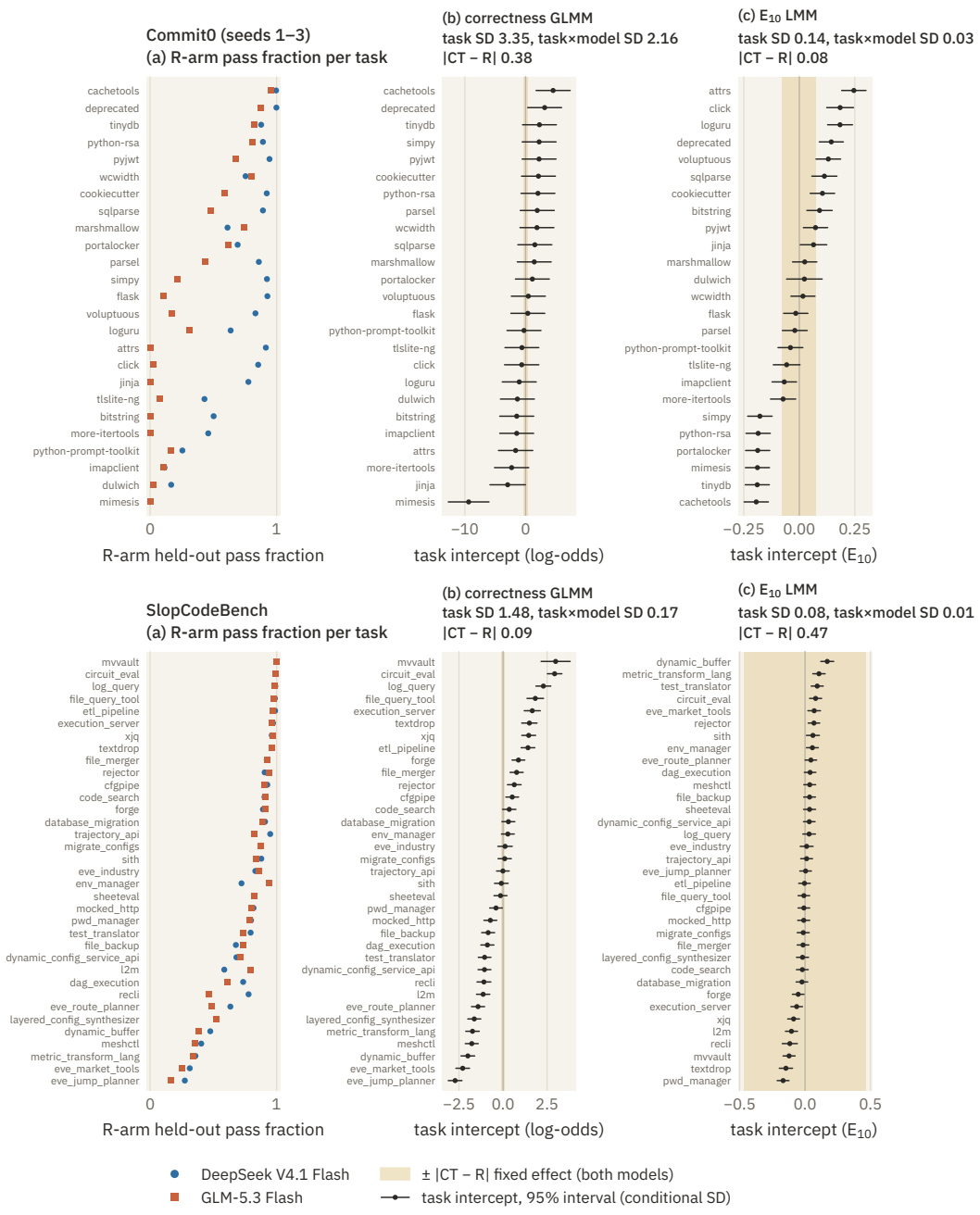

Figure 7: Between-task variability dominates correctness but not erosion. Top Commit0 (seeds 1–3, 25 libraries), bottom SlopCodeBench (35 problems). (a) Pass fraction of the reminder-only arm per task and model. (b) Task random intercepts of the correctness GLMM (log-odds) and (c) of the E_{10} LMM, sorted, with 95% intervals from the conditional standard deviations; the shaded band is $\pm$ the CT–R fixed effect (average over the two models) on the same scale. On correctness the task standard deviation is 3.35 log-odds on Commit0 and 1.48 on SlopCodeBench, against a CT–R effect of 0.38 and 0.09; on SlopCodeBench E_{10} the CT–R effect (0.47) is about six times the task standard deviation (0.078).

Table 13: Sensitivity of the primary contrast CT–R. Commit0: pooled over models (post hoc), pass fraction in points at seeds 1–3 and 1–5, and E_{10} at seeds 1–5. SCB: pass fraction in points for each model and DeepSeek E_{10}. Paired 95% t intervals (pooled: normal). Blank: the rule does not apply to that benchmark. "Adverse": a missing C/T/CT cell takes its problem's worst included value, a missing B0/R cell the best. "Scored 0" is an extreme bound (its GLMM does not converge).

Variant	C0 pass 1–3	C0 pass 1–5	C0 E_{10}	SCB DS pass	SCB GLM pass	SCB DS E_{10}
As analyzed	+4.6 [+0.6, +8.5]	+2.7 [-0.2, +5.6]	−0.080 [-0.110, -0.051]	−2.4 [-6.1, +1.2]	+0.0 [-3.1, +3.2]	−0.433 [-0.487, -0.379]
No-report zeros dropped	+2.1 [-0.5, +4.7]	+0.1 [-2.2, +2.3]	−0.085 [-0.117, -0.053]			
Timeout/kill zeros dropped	+3.5 [+0.3, +6.7]	+1.1 [-1.3, +3.5]	−0.079 [-0.109, -0.049]			
dulwich, bitstring, jinja dropped	+3.9 [-0.4, +8.2]	+2.6 [-0.5, +5.8]	−0.081 [-0.114, -0.049]			
E_{10} from last measured rung	+4.6 [+0.6, +8.5]	+2.7 [-0.2, +5.6]	−0.079 [-0.109, -0.050]			
Set-aside cells put back	+3.8 [-0.1, +7.6]	+1.9 [-0.9, +4.7]	−0.080 [-0.109, -0.051]			
Complete blocks only	+4.0 [-0.8, +8.8]	+3.3 [-0.6, +7.1]	−0.091 [-0.124, -0.057]	−1.3 [-4.7, +2.2]	−0.6 [-3.5, +2.2]	−0.421 [-0.475, -0.367]
eve_market_tools/R/s1 included				−2.4 [-6.1, +1.2]	+0.2 [-3.1, +3.5]	−0.433 [-0.487, -0.379]
T/CT cells rerun, fixed harness (post hoc)				−2.0 [-5.0, +0.9]	−0.0 [-2.9, +2.8]	−0.419 [-0.473, -0.365]
Missing cells adverse to C/T/CT				−2.9 [-6.2, +0.3]	−0.7 [-4.1, +2.7]	−0.313 [-0.393, -0.233]
Missing cells scored 0				−13.0 [-23.4, -2.6]	−4.5 [-13.4, +4.3]	−0.433 [-0.487, -0.379]

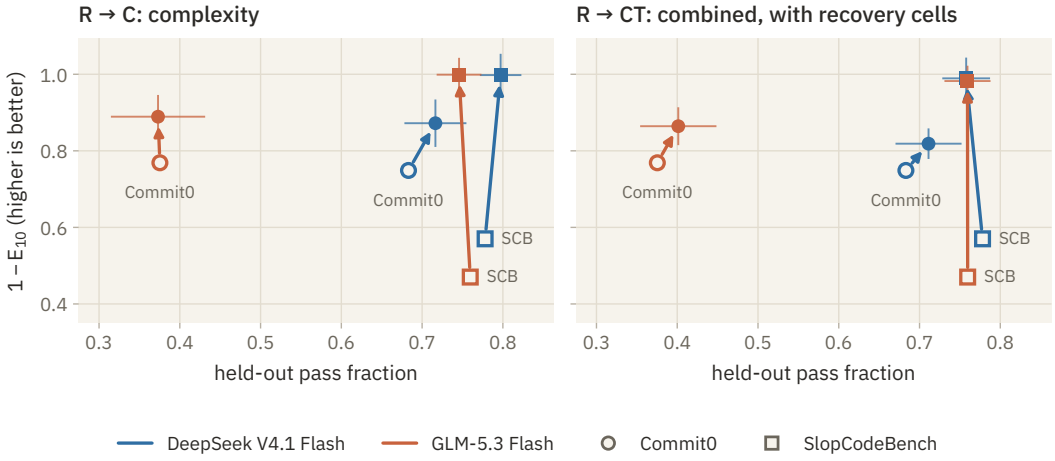

Figure 8: Post hoc variant of the R→C and R→CT arrows (Figures 1(b) and 5(a)): SlopCodeBench CT uses the 40 T/CT cells rerun on the fixed harness (16 GB memory) in place of the missing ones; Commit0 and SlopCodeBench C are unchanged. The complete-case figure remains the primary analysis.

L RESPONSE TO FEEDBACK

Did the agents act on the messages? An audit of every included cell answers this from the turn records. It follows each function a message named, and each over-limit or under-covered function the agent edited whether or not it was named, from the flag to the end of the run. The audit reproduces the Commit0 quantities of the original analysis exactly (3,002 of 3,002 cell-level values match), and 40 sampled flags were recomputed independently. Delivery is not the issue: in every benchmark, model and arm, the agent saw at least 98.2% of the messages the harness offered.

Complexity. Named functions are acted on, and mostly fixed. On SlopCodeBench, agents edit 82–94% of the functions the complexity table names again, a median of two to three turns later, and 82–94% of named functions end the run under the limit. Under R, over-limit functions the agent happens to edit end under the limit in 4–5% of cases. Commit0 shows the same contrast at a lower level: 36–60% of named functions are edited again and 23–58% end under the limit, against 1–3% under R (Table 4). The response is weaker under CT than under C on both benchmarks, as if the second table competed for the agent's attention.

Coverage. To compare arms fairly, the table uses the audit's broader population: every function an agent edited while its coverage was below 1, in every arm. On SlopCodeBench, coverage of such functions later rises in 92–95% of cases under T and CT with DeepSeek, against 17% under R; with GLM, in 38–63% against 10%. On Commit0 the uncovered-lines table makes no difference: coverage rises in 57% of cases (T, CT) against 64% (R) with DeepSeek, and 34–42% against 40% with GLM. On Commit0, then, coverage feedback changes neither coverage nor structure.

M EXPLORATORY MODEL: QWEN 3.8 27B

On five SlopCodeBench problems with two seeds, complexity feedback lowers E_{10} by 0.53 (C–R $[-0.69, -0.37]$) and the combined feedback by 0.52 (CT–R $[-0.66, -0.37]$). No correctness contrast is resolved: CT–R is -5.1 points $[-16.1, 5.8]$, with an odds ratio of 0.68 [0.40, 1.18]. Five problems support no more than a direction.

N TOKEN AND STEP EFFICIENCY (EXPLORATORY)

We asked whether complexity feedback made the agent reach its result with fewer tokens or fewer steps on Commit0; this analysis was not pre-registered. Across the 25 libraries it did not, at least not relative to the reminder-only control (Table 14, top). C and CT use about as many processed tokens and agent steps per cell as R under both models. Every ratio lies between 0.94 and 1.11, and every interval includes 1 except one: under GLM, CT uses 11% more tokens than R, an interval that excludes 1 before rounding ([1.005, 1.225]). Relative to B0, C uses 13–17% fewer tokens (DeepSeek 0.83 [0.70, 0.98], GLM 0.87 [0.77, 0.99]). Since C sits at R's level, that saving belongs to the reminder the feedback arms share, not to the complexity table. Under DeepSeek, C and CT also take 1.6–1.7 times as many steps as B0: the saving comes from shorter steps, not fewer.

Individual libraries do show clear gains. The clearest is GLM on parsel under C, which reaches twice R's pass fraction (0.87 against 0.43, and 0.34 under B0) with about 40% fewer tokens (1.01M against 1.75M and 1.85M). All three seeds reach R's final pass fraction by 400k tokens and stop on their own, whereas R reaches it in one seed of three. DeepSeek on simpy under C matches both controls' pass fraction with 0.73M tokens against 0.91M and 1.60M (Table 14, bottom). Such cases are not evidence of an efficiency effect, however. A library counts as a gain when its arm is within one point of the better control's pass fraction and either uses at least 20% fewer tokens or steps than each control, or reaches R's mean final pass fraction at a lower budget than both. Under these rules, a feedback arm beats both controls on 26 model×library pairs and a control beats both feedback arms on 23, which is what chance alone would produce.

Table 14: Commit0 token and step use, seeds 1–3 (exploratory, not pre-registered). Top: geometric-mean ratio of processed tokens and agent steps per cell, feedback arm over control, paired over the 25 libraries, with 95% t intervals on the log scale. Bottom: the five clearest libraries where a complexity arm beat both controls, shown as arm / R / B0; the last column counts the seeds that reached R's mean final pass fraction within the 2M cap. Under the same rules, a feedback arm beats both controls on 26 model×library pairs and R or B0 beats both feedback arms on 23.

		vs R		vs B0	
Model	Arm	tokens	steps	tokens	steps
DeepSeek	C	0.94 [0.86, 1.04]	1.04 [0.93, 1.16]	0.83 [0.70, 0.98]	1.70 [1.45, 2.00]
DeepSeek	CT	0.94 [0.83, 1.07]	0.95 [0.86, 1.04]	0.83 [0.69, 1.00]	1.56 [1.35, 1.80]
GLM	C	0.98 [0.86, 1.12]	0.99 [0.89, 1.11]	0.87 [0.77, 0.99]	1.01 [0.90, 1.14]
GLM	CT	1.11 [1.00, 1.23]	1.01 [0.92, 1.12]	0.99 [0.93, 1.05]	1.03 [0.92, 1.16]

Model	Library	Arm	Pass	Tokens (M)	Steps	Reached R
GLM	parsel	C	0.87 / 0.43 / 0.34	1.01 / 1.75 / 1.85	109 / 104 / 82	3/3, 1/3, 2/3
DeepSeek	simpy	C	0.95 / 0.92 / 0.95	0.73 / 0.91 / 1.60	148 / 146 / 108	3/3, 2/3, 3/3
DeepSeek	cachetools	CT	1.00 / 1.00 / 1.00	0.18 / 0.53 / 0.46	63 / 74 / 42	2/3, 2/2, 3/3
GLM	simpy	C	0.78 / 0.22 / 0.75	1.24 / 1.85 / 1.81	126 / 143 / 145	3/3, 3/3, 3/3
GLM	voluptuous	C	0.37 / 0.17 / 0.26	1.95 / 1.78 / 1.69	88 / 115 / 123	2/3, 2/3, 1/3

O ILLUSTRATIVE TRACE EXCERPTS

Each excerpt below follows one feedback message through to its outcome, on a real cell. It opens with a **Takeaway**: what the case shows, with every claim checked against the run's data. A **measured** table follows, giving the numbers and the files and commits they come from; each complexity value is marked **within** or **over** the limit of 6. Next come the flagged code **before** and **after**, and finally the step-by-step timeline: SlapKit's message (orange), the agent's tool calls (blue) with their results (white), up to the measurement that settles the case. The models' private reasoning is never shown. Where a reasoning step occurred, only its length is given. Verbatim lines are copied from the run; titles, labels and takeaways are ours.

There are two excerpts per benchmark for each of six situations: the reminder alone (R), the coverage table (T), and complexity feedback under C and CT, where the agent either split the function into meaningful parts or introduced trivial wrappers. The six carrying a green "recommended for the final appendix" label are our suggestion for a shorter final appendix: Commit0 jinja (C), voluptuous (CT) and pyjwt (T); SlopCodeBench circuit_eval (C), mocked_http (C, trivial wrappers) and database_migration (R).

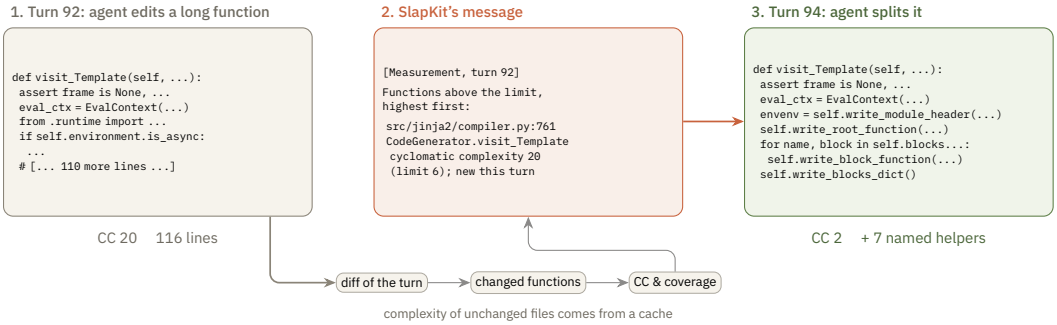

Figure 9: How to read an excerpt, on its shortest case (Commit0, DeepSeek, jinja, C arm). The agent edits a long function (grey, CC 20). SlapKit measures the functions the turn changed and sends its message (orange). Two turns later the function is split into named helpers (green, CC 2). Each excerpt below adds the measured table and the full timeline; this one is *C-meaningful-1* in the Commit0 list.

O.1 COMMIT0 EXCERPTS

COMMIT0 / GLM-5.3 FLASH / PYJWT / T: CLOSING TWO UNCOVERED LINES WITH A DIRECT TEST

Cell `commit0/glm/pyjwt/T/s3`, turns 10–11 of 52. `base64url_encode` strips base64 padding for JWT's URL-safe encoding.

Takeaway

The table names two lines in `base64url_encode` (23–24, its entire two-line body) that no test executed; no test loaded `jwt/utils.py` at all. The agent responds with a dedicated test module, and the test shown exercises the padding-stripping path. The function's coverage goes from 0% to 100% at the next measured turn: every line the table named is now covered.

measured complexity — coverage of the flagged lines, limit 0 uncovered

function	flagged uncovered lines	cov turn 10	cov turn 11
`base64url_encode`	23, 24	0%	100% **within**

Where to check: `pilot4-glm-5seed/results/pyjwt/T/s3/run/measurements/turn-NNN-functions.tsv`, column `cov`, for turns 10, 11 (function `jwt/utils.py`); uncovered line numbers from that turn's delivered packet; code in the run's snapshot commits (turn 10 `5acd786695`, turn 11 `6be91da752`).

code before — turn 10 · 2 uncovered line(s): no test executes them

```
# -- base64url_encode --
def base64url_encode(input: Union[bytes, str]) -> str:
return base64.urlsafe_b64encode(force_bytes(input)).decode('ascii').rstrip('=')
```

code after — turn 11 · the test written in response

```
def test_base64url_encode_strips_padding():
    encoded = base64url_encode(b"x")
    assert "=" not in encoded
    assert encoded == "eA"
```

How it happened (turns 10–11 of 52; the fixed instructions that close every SlapKit message are omitted, see Appendix C)

SlapKit feedback | agent tool call | tool result | code before | code after

SlapKit feedback — turn 10 · 887k tokens

```
Functions above the limit, highest first:
  jwt/utils.py:23  base64url_encode
    uncovered lines 2 (limit 0): no test loads this file; was 2 at turn 1
At or under the limit: 0 functions.
```

agent — after turn 10

```
> bash: sleep 2; cat /tmp/bgout 2>/dev/null; echo retry
> write: tests/test_utils.py
> edit: tests/test_utils.py
> read: tests/test_utils.py:56-95
> edit: tests/test_utils.py
> write: tests/test_algorithms.py
```

SlapKit feedback — turn 11 · 910k tokens

no packet delivered at turn 11 (0 functions touched this turn)

COMMIT0 / DEEPSEEK V4.1 FLASH / JINJA / C: A COMPILER PASS SPLIT INTO SEVEN NAMED HELPERS, ONE LEFT OVER THE LIMIT

Cell `commit0/deepseek/jinja/C/s3`, turns 92–95 of 181. `visit_Template` writes the Python source of a compiled Jinja template.

Takeaway

The flag worked for the function it named: in one edit, `visit_Template` fell from CC 20, well over the limit of 6, to CC 2, split into seven named helpers. But some complexity moved rather than disappeared. One helper, `write_module_header`, is itself over the limit (CC 8). SlapKit flagged it once, at turn 94, and the agent never edited it again; because SlapKit reports only functions changed in the current turn, it never flagged the helper again either, and the helper is still at CC 8 at the last turn (181). The table below gives every number and its source.

measured complexity — cyclomatic complexity (CC), limit 6

function	turn 92	turn 94	turn 181
`visit_Template`	20 **over**	2 **within**	2 **within**
`write_module_header`	–	8 **over**	8 **over**
`write_import_alias`	–	3 **within**	3 **within**
`write_root_function`	–	4 **within**	4 **within**
`write_parent_render`	–	2 **within**	2 **within**
`write_block_function`	–	2 **within**	2 **within**
`write_block_locals`	–	3 **within**	3 **within**
`write_blocks_dict`	–	3 **within**	3 **within**
sum over these functions	20	27	27

Where to check: `pilot4-deepseek-5seed/results/jinja/C/s3/run/measurements/turn-NNN-functions.tsv`, column `ccn`, for turns 92, 94, 181; code in the run's snapshot commits (turn 92 `7b1fedf313`, turn 94 `6802badaec`, turn 181 `f42fe51a65`).

code before — turn 92 · CC 20: over the limit of 6

```
def visit_Template(self, node: nodes.Template, frame: t.Optional[Frame]=None) -> None:
    assert frame is None, 'no root frame allowed'
    eval_ctx = EvalContext(self.environment, self.name)
    from .runtime import async_exported
    from .runtime import exported
    if self.environment.is_async:
        # [...]
    # generate the root render function.
    self.writeline(f"{self.func('root')}(context, missing=missing{envenv}):")
    self.indent()
    self.write_commons()
# [...]
```

code after — turn 94 · CC 2: within the limit

```
def visit_Template(self, node: nodes.Template, frame: t.Optional[Frame]=None) -> None:
    assert frame is None, 'no root frame allowed'
    eval_ctx = EvalContext(self.environment, self.name)
    envenv = self.write_module_header(node)

    # generate the root render function.
    self.write_root_function(node, eval_ctx, envenv)

    # at this point we now have the blocks collected and can visit them too.
    for name, block in self.blocks.items():
        self.write_block_function(name, block, eval_ctx, envenv)

    self.write_blocks_dict()
```

How it happened (turns 92–95 of 181; the fixed instructions that close every SlapKit message are omitted, see Appendix C)

SlapKit feedback | agent tool call | tool result | code before | code after

SlapKit feedback — turn 92 · 1,316k tokens

```
Functions above the limit, highest first:
  src/jinja2/compiler.py:761  CodeGenerator.visit_Template
    cyclomatic complexity 20 (limit 6); new this turn
At or under the limit: 0 functions.
```

agent — after turn 92

```
> bash: cd /results/run/tree && /bench/open/venv/jinja/bin/python -m pytest tests -q 2>&1 |
  tail -40
```

SlapKit feedback — turn 93 · 1,349k tokens

```
  [...]
At or under the limit: 0 functions.
```

agent — after turn 93

a private reasoning step of about 9,000 characters precedes this edit (not shown)

```
> edit: src/jinja2/compiler.py  (replaces 5,305 characters with 6,520)
```

```
761:    def visit_Template(self, node: nodes.Template, frame: t.Optional[Frame]= ...
```

```
> edit: src/jinja2/parser.py  (replaces 518 characters with 518)
```

SlapKit feedback — turn 94 · 1,357k tokens

```
  src/jinja2/compiler.py:775  CodeGenerator.write_module_header
    cyclomatic complexity 8 (limit 6); new this turn
At or under the limit: 7 functions.
```

agent — after turn 94

```
> edit: src/jinja2/compiler.py  (replaces 255 characters with 289)
```

SlapKit feedback — turn 95 · 1,365k tokens

```
none
At or under the limit: 1 functions.
```

COMMIT0 / GLM-5.3 FLASH / VOLUPTUOUS / CT: ONE DISPATCHER, THREE NAMED VALIDATION STRATEGIES, ONE LEFT OVER THE LIMIT

Cell `commit0/glm/voluptuous/CT/s5`, turns 15–18 of 64. `_run` branched on the validator's own class (`All`, `SomeOf`, else) with duplicated invoke/except logic in each branch.

Takeaway

`_run` split cleanly into three named validation strategies plus a shared invoke helper (CC 18 to CC 6), yet the story repeats the jinja case almost exactly. One new helper, `_run_someof`, is itself CC 7, one over the limit. SlapKit flagged it once, at turn 17, the very turn this response landed; the agent never edited it again, and it is still CC 7 at the last turn (64). Two cases now show the same shape: a decomposition that satisfies the top-level flag can still leave one helper over the limit, silently, because the flag only looks at the current turn's edits.

measured complexity — cyclomatic complexity (CC), limit 6

function	turn 15	turn 17	turn 64
`_run`	18 **over**	6 **within**	6 **within**
`_invoke`	–	2 **within**	1 **within**
`_run_any`	–	6 **within**	6 **within**
`_run_all`	–	4 **within**	4 **within**
`_run_someof`	–	7 **over**	7 **over**
sum over these functions	18	25	24

Where to check: `pilot4-glm-5seed/results/voluptuous/CT/s5/run/measurements/turn-NNN-functions.tsv`, column `ccn`, for turns 15, 17, 64; code in the run's snapshot commits (turn 15 `567fa027d8`, turn 17 `1d37673793`, turn 64 `edb85b9fdc`).

code before — turn 15 · CC 18: over the limit of 6

```
def _run(self, path, v):
    if self.discriminant is not None:
        filtered = self.discriminant(v, self.validators)
        validators = [self._compiled[self.validators.index(x)] for x in filtered]
    else:
        validators = self._compiled
    if isinstance(self, All):
        res = v
        for validate in validators:
            try:
                # [...]
            return validate(path, v)
        # [...]
```

code after — turn 17 · CC 6: within the limit

```
def _run(self, path, v):
    """Compiled mode: apply discriminant filtering and dispatch."""
    validators = self._compiled
    if self.discriminant is not None:
        chosen = list(self.discriminant(v, self.validators))
        validators = [c for val, c in zip(self.validators, self._compiled) if val in chosen]
    if isinstance(self, All):
        return self._run_all(validators, path, v)
    if isinstance(self, SomeOf):
        return self._run_someof(validators, path, v)
    return self._run_any(validators, path, v)
```

How it happened (turns 15–18 of 64; the fixed instructions that close every SlapKit message are omitted, see Appendix C)

SlapKit feedback | agent tool call | tool result | code before | code after

SlapKit feedback — turn 15 · 1,198k tokens

```
Functions above the limit, highest first:
  voluptuous/validators.py:197  _WithSubValidators._run
    cyclomatic complexity 18 (limit 6); new this turn
  [...]
At or under the limit: 4 functions.
Uncovered lines counts, for each function you changed this [...]
Functions above the limit, highest first:
  voluptuous/validators.py:197  _WithSubValidators._run
    uncovered lines 37 (limit 0): no test loads this file; new this turn
  [...]
At or under the limit: 0 functions.
```

agent — after turn 15

```
> bash: /bench/open/venv/voluptuous/bin/python -W ignore - «'EOF' 2>&1  [... 30 more line(s
  ) ...]

> read: voluptuous/validators.py:1-13

> edit: voluptuous/validators.py
```

SlapKit feedback — turn 16 · 1,220k tokens

no packet delivered at turn 16 (0 functions touched this turn)

agent — after turn 16

```
> read: voluptuous/schema_builder.py:156-420

> edit: voluptuous/schema_builder.py
```

SlapKit feedback — turn 17 · 1,273k tokens

```
  voluptuous/validators.py:237  _WithSubValidators._run_someof
    cyclomatic complexity 7 (limit 6); new this turn
At or under the limit: 5 functions.
Uncovered lines counts, for each function you changed this [...]
Functions above the limit, highest first:
  voluptuous/validators.py:237  _WithSubValidators._run_someof
    uncovered lines 14 (limit 0): no test loads this file; new this turn
  [...]
  voluptuous/validators.py:201  _WithSubValidators._run
    uncovered lines 7 (limit 0): no test loads this file; was 37 at turn 15
  [...]
At or under the limit: 0 functions.
```

agent — after turn 17

```
> read: voluptuous/validators.py:27-150

> edit: voluptuous/validators.py

> read: voluptuous/schema_builder.py:876-920
```

SlapKit feedback — turn 18 · 1,298k tokens

```
none
At or under the limit: 5 functions.
Uncovered lines counts, for each function you changed this [...]
Functions above the limit, highest first:
  [...]
At or under the limit: 0 functions.
```

SLOPCODEBENCH / GLM-5.3 FLASH / DATABASE_MIGRATION / R: CONTINUING THE CHECKPOINT INSTEAD OF REVIEWING, AND A BUG THAT OUTLIVES THE TURN BY ONE

Cell `scb/glm/database_migration/R/s2`, checkpoint 4, turns 65–67. `_operation_event` builds the JSON payload logged for one applied or rolled-back schema-migration operation.

Takeaway

R carries no complexity or coverage table, only the fixed readability reminder, and the turn's own visible text says the agent will keep implementing rather than pause for it: it goes on adding a rollback feature. Its edit gives `_operation_event` a new `event_name` parameter (lines 1232–1233 build a dict using it), then, two lines later, silently overwrites both variables back to the hard-coded string `"operation_applied"` (lines 1234–1235): the new parameter is built and immediately discarded. `_operation_event`'s CC (10, already well over the limit) and coverage (0%, untested either turn) are untouched by all this, so nothing in this packet's own numbers would have shown the defect either. The duplicate lines are gone by the very next measured turn, 67, one checkpoint-relative reminder later and still with no CC or coverage table naming this function; the fix, whatever prompted it, was not a response to a SlapKit table.

measured complexity — what the reminder did not catch, limit 6

	turn 65	turn 66	turn 67
`_operation_event` CC	10 **over**	10 **over**	10 **over**
`event_name` honored?	n/a	**over** no (ln 1235)	**within** yes

Where to check: `scb-fenced-glm53-2seeds-recovery-20260923/results/scb/database_migration/R/s2/scb-ccff4bb31ac0c65f22f8c201/results/scb/database_migration/R/s2/run/measurements/turn-065-functions.tsv` through `turn-067-functions.tsv`, column `ccn`; source at the run's snapshot commits (turn 65 `b1cbb0e1f3`, turn 66 `f7f1fa6696`, turn 67 `e3e6004c7d`), lines 1232–1235 of `migration_tool.py`.

code before — turn 65 · CC 10: the reminder carries no data about this function

```
def _operation_event(op, version):
    """Build the operation_applied event payload for one applied operation.

    Key order matches the specification examples: index events list 'name'
    before 'table'; foreign key and check constraint events list 'table'
    before 'name'.
    """
    op_type = op["type"]
    event = {"event": "operation_applied", "type": op_type}
    if op_type in NAME_FIRST_OPS:
        event["name"] = op["name"]
    if "table" in op:  # drop_index is database-scoped and has no table
    # [...]
    event["version"] = version
    return event
```

code after — turn 66 · CC 10: what changed anyway

```
def _operation_event(op, version, event_name="operation_applied"):
    """Build the event payload for one applied or rolled-back operation.

    Key order matches the specification examples: index events list 'name'
    before 'table'; foreign key and check constraint events list 'table'
    before 'name'.
    """
    op_type = op["type"]
    event = {"event": event_name, "type": op_type}
    op_type = op["type"]
    event = {"event": "operation_applied", "type": op_type}
    if op_type in NAME_FIRST_OPS:
        event["name"] = op["name"]
    if "table" in op:  # drop_index is database-scoped and has no table
    # [...]
    event["version"] = version
    return event
```

How it happened (turns 65–66)

SlapKit feedback | agent tool call | tool result | code before | code after

SlapKit feedback — turn 65 · 130k tokens

```
[standard instructions, identical in every packet; see the Prompts appendix]
```

agent — after turn 65

a private reasoning step of about 600 characters precedes this (not shown)

```
While that runs, I'll continue implementing. Reading the current 'apply_migration'/'_opera
  tion_event'/'main' regions after the renumbering:

> grep: migration_tool.py
```

a private reasoning step of about 466 characters precedes this (not shown)

```
Expected: 'run_migrate' doesn't yet populate 'operations'. Now the core edits — metadata
  capture in 'apply_migration', event refactor, rollback engine, CLI dispatch:

[... further tool calls/text in this window elided; see the takeaway for what they were ..
  .]
```

SLOPCODEBENCH / GLM-5.3 FLASH / CIRCUIT_EVAL / C: A TOPOLOGICAL EVALUATOR SPLIT INTO PLAN AND DRAIN

Cell `scb/glm/circuit_eval/C/s1`, checkpoint 2, turns 17–19. `evaluate_circuit` runs a logic-gate netlist to a fixed point, a Kahn's-algorithm-style evaluation over the gate graph.

Takeaway

The flag worked, and kept working well past the one-turn window this excerpt shows: CC 10 → 1 the same turn, held through the rest of the checkpoint (turn 31) and to the cell's last measured turn, 129, 110 measured turns later. Both new helpers, `build_eval_plan` and `drain_eval_queue`, are substantive (CC 5 and 6, 11 lines each) and stay at exactly those numbers throughout. This is a real split rather than a relabeling: building the plan and draining the queue are distinct parts of the algorithm. The agent's own text opens by saying a previous edit left the `dependents` tracking in the wrong place and broke the file; at turn 17 that tracking is already a local variable inside `evaluate_circuit` itself, the same one later threaded through both new helpers. The agent re-reads that exact span before changing it (edit not shown), so this is a repair inside the flagged function, not an unrelated detour.

measured complexity — cyclomatic complexity (CC), limit 6

function	turn 17	turn 19	turn 31	turn 129
`evaluate_circuit`	10 **over**	1 **within**	1 **within**	1 **within**
`build_eval_plan`	–	5 **within**	5 **within**	5 **within**
`drain_eval_queue`	–	6 **within**	6 **within**	6 **within**
sum over these functions	10	12	12	12

Where to check: `scb-fenced-glm53-2seeds-20260923/results/scb/circuit_eval/C/s1/scb-ccff4bb31ac0c65f22f8c201/results/scb/circuit_eval/C/s1/run/measurements/turn-NNN-functions.tsv`, column `ccn`, for turns 17, 19, 31, 129; code in the run's snapshot commits (turn 17 `7e02b6ba48`, turn 19 `89d8a3ac9a`, turn 31 `921239fc23`, turn 129 `1e4c8e7df7`).

code before — turn 17 · CC 10: over the limit of 6

```
def evaluate_circuit(circ, input_values):
    """Evaluate all assignments in dependency order; returns name -> 0/1."""
    assignments = {a.lhs: a for a in circ.assignments}
    remaining = {}
    dependents = {name: [] for name in assignments}
    # [...]
                queue.append(succ)
    return values
```

code after — turn 19 · CC 1: within the limit of 6

```
def evaluate_circuit(circ, input_values):
    """Evaluate all assignments in dependency order; returns name -> 0/1."""
    assignments, remaining, dependents = build_eval_plan(circ)
    values = dict(input_values)
    return drain_eval_queue(assignments, remaining, dependents, values)
```

How it happened (turns 17–19)

SlapKit feedback | agent tool call | tool result | code before | code after

SlapKit feedback — turn 17 · 73k tokens

```
Functions above the limit, highest first:

  circopt.py:505  evaluate_circuit
    cyclomatic complexity 10 (limit 6); new this turn

At or under the limit: 1 functions.

[standard instructions, identical in every packet; see the Prompts appendix]
```

agent — after turn 17

a private reasoning step of about 2,527 characters precedes this (not shown)

```
The measurement flags complexity, but first my last edit landed the ‘dependents‘ line in t
  he wrong place (broke the file). Let me read the section and repair properly:

> read: /work/cell/tree/circopt.py:478-527

[... further tool calls/text in this window elided; see the takeaway for what they were ..
  .]
```

SlapKit feedback — turn 19 · 101k tokens · resolved

```
Functions above the limit, highest first:

none

At or under the limit: 3 functions.

[standard instructions, identical in every packet; see the Prompts appendix]
```

SLOPCODEBENCH / GLM-5.3 FLASH / MOCKED_HTTP / C: A NUMERIC-FORMAT DISPATCH TABLE

Cell `scb/glm/mocked_http/C/s2`, checkpoint 1, turns 1–3. `_format_numeric` renders one recorded value under a verb such as `int`, `fixed` or `hex`.

Takeaway

The CC drop is real (8 → 2), but the six new helpers it relies on are mostly not. Three of them, `_fmt_int`, `_fmt_general` and `_fmt_hex_upper`, are single-line bodies with CC 1 and 2 lines of code each, trivial by the wrappers audit's own rule (CC 1, NLOC ≤ 3). The other three (`_fmt_fixed`, `_fmt_exp`, `_fmt_hex`) do carry a branch (CC 2) but are still two lines long. All six are reached only through this dispatch mechanism, though not each exactly once: the new `_NUMERIC_VERBS` table maps two verb characters each to `_fmt_fixed` (`f`/`F`) and to `_fmt_general` (`g`/`G`), hence 2 refs for each in the table below, and `_fmt_hex` also shows 2: one table entry (`x`) plus one direct call from `_fmt_hex_upper`, which builds on it rather than duplicating its body. The packet names 14 over-limit functions in the same file at once; the row shown below is the one this excerpt follows, and the rest are elided.

measured complexity — cyclomatic complexity (CC), limit 6

function	turn 1	turn 3	turn 28	turn 121	NLOC, turn 3	refs
_format_numeric	8 over	2 within	2 within	2 within	5	1
_fmt_int	–	1 within	1 within	1 within	2	1
_fmt_fixed	–	2 within	2 within	2 within	2	2
_fmt_exp	–	2 within	2 within	2 within	2	1
_fmt_general	–	1 within	1 within	1 within	2	2
_fmt_hex	–	2 within	2 within	2 within	4	2
_fmt_hex_upper	–	1 within	1 within	1 within	2	1
sum over these functions	8	11	11	11		

Where to check: `scb-fenced-glm53-2seeds-20260923/results/scb/mocked_http/C/s2/scb-ccff4bb31ac0c65f22f8c201/results/scb/mocked_http/C/s2/run/measurements/turn-NNN-functions.tsv`, column `ccn`, for turns 1, 3, 28, 121; code in the run's snapshot commits (turn 1 `5f57c76975`, turn 3 `2eef97b455`, turn 28 `66afe5cf64`, turn 121 `5b733fe7c6`). `refs` is every whole-word occurrence of the helper's name in the file at turn 3 outside its own `def` line – a helper reached from two dispatch-table entries, or called again from another new helper, counts more than once.

code before — turn 1 · CC 8: over the limit of 6

```
def _format_numeric(verb, prec, arg):
    if verb == 'd':
        return str(int(arg))
    if verb in ('f', 'F'):
        return '%.*f' % (int(prec or 6), _to_float(arg))
    if verb == 'e':
        return '%.*e' % (int(prec or 6), _to_float(arg))
    if verb in ('g', 'G'):
        return '%g' % _to_float(arg)
    if verb in ('x', 'X'):
        return _format_hex(verb, arg)
    raise TemplateError('unsupported verb %%%s' % verb)
```

code after — turn 3 · CC 2: within the limit of 6

```
def _format_numeric(verb, prec, arg):
    handler = _NUMERIC_VERBS.get(verb)
    if handler is None:
        raise TemplateError('unsupported verb %%%s' % verb)
    return handler(prec, arg)
```

How it happened (turns 1–3)

SlapKit feedback | agent tool call | tool result | code before | code after

SlapKit feedback — turn 1 · 143k tokens

```
Functions above the limit, highest first:

  [...]
  hmock.py:105  _format_numeric
    cyclomatic complexity 8 (limit 6); new this turn
  [...]

At or under the limit: 137 functions.

[standard instructions, identical in every packet; see the Prompts appendix]
```

agent — after turn 1

a private reasoning step of about 12,915 characters precedes this (not shown)

```
Refactoring the flagged functions to meet the complexity target, then testing. I'll rewrit
  e the file with the decomposed helpers (many scattered refactors — full rewrite is clea
  ner).

> write: hmock.py

[... further tool calls/text in this window elided; see the takeaway for what they were ..
  .]
```

SlapKit feedback — turn 3 · 222k tokens · resolved

```
Functions above the limit, highest first:

  [...]

At or under the limit: 45 functions.

[standard instructions, identical in every packet; see the Prompts appendix]
```