UNTILITPASSES: BENCHMARKING REPOSITORY GENERATION AGAINST A VERIFIABLE GOAL

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic coding capabilities have expanded from resolving individual issues to generating entire repositories. Existing repository benchmarks often evaluate agents in a single run and identify premature exit and unhandled edge cases as the leading failure patterns. However, real-world agentic coding involves feedback and multiple rounds of revision. To bridge this gap, we introduce UntilItPasses, an evaluation of 46 challenging repository-construction tasks that replaces the single submission with a minimal feedback loop. Each agent builds incrementally against a natural-language specification, commits its changes throughout development, and may make a feedback call at any point. Feedback reports only the fraction of hidden tests that pass, providing a milestone signal without revealing individual test cases. The agent continues working until it receives a perfect result or reaches a calibrated budget equivalent to six hours. Across twelve agent systems, 2.5% of runs exit on a perfect score. Our analysis of commit histories and feedback-call records traces agents' progress through initial implementation and later repair, showing rapid early gains and persistent implementation gaps. Selected trajectories show agents using scalar feedback to broaden self-tests and revise mistaken assumptions shared by their code and tests. Model efficiency rankings also change with the required completion level. These findings highlight feedback-guided self-correction as a key capability to develop and evaluate in autonomous coding agents.

1 INTRODUCTION

Large language models (LLMs) are increasingly used as coding agents to tackle complex, long-horizon software engineering tasks. Recent benchmarks study these tasks through the construction of complete repositories. Commit0 (Zhao et al., 2024) requires agents to implement interdependent library APIs from specifications and starter code. NL2Repo-Bench evaluates the construction of complete libraries from natural-language requirements, including architectural design, dependency management, implementation, and packaging (Ding et al., 2025). These tasks require agents to decompose a development goal and coordinate behavior across modules through successive changes.

Evaluations of long-horizon software engineering report recurring failure patterns that include premature termination and missed edge cases. NL2Repo-Bench describes agents that claim completion prematurely or halt to await further user guidance, leaving repositories unfinished (Ding et al., 2025). SWE-bench Pro explicitly includes missed edge cases in its taxonomy of failed agent trajectories (Deng et al., 2025). SWE-Marathon identifies premature termination and poor self-verification, including cases where local checks pass but broader verification exposes unsupported configurations or unhandled negative cases (Desai et al., 2026). These observations highlight the difficulty of satisfying the full task in a single run, yet real-world development proceeds through iterative feedback and revision. This motivates a central question: *Can coding agents sustain progress toward a verifiable goal in repository generation?*

To study this question, we introduce UntilItPasses, a benchmark and protocol for goal-directed repository generation. Each task provides a natural-language specification and a workspace with the implementation removed. The agent develops the library incrementally and may request workspace feedback at any point. Each response reports only the fraction of hidden tests that the current workspace passes, with no diagnostic detail. The protocol requires the agent to continue development until it achieves verified completion or exhausts its budget. The agent commits changes throughout

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

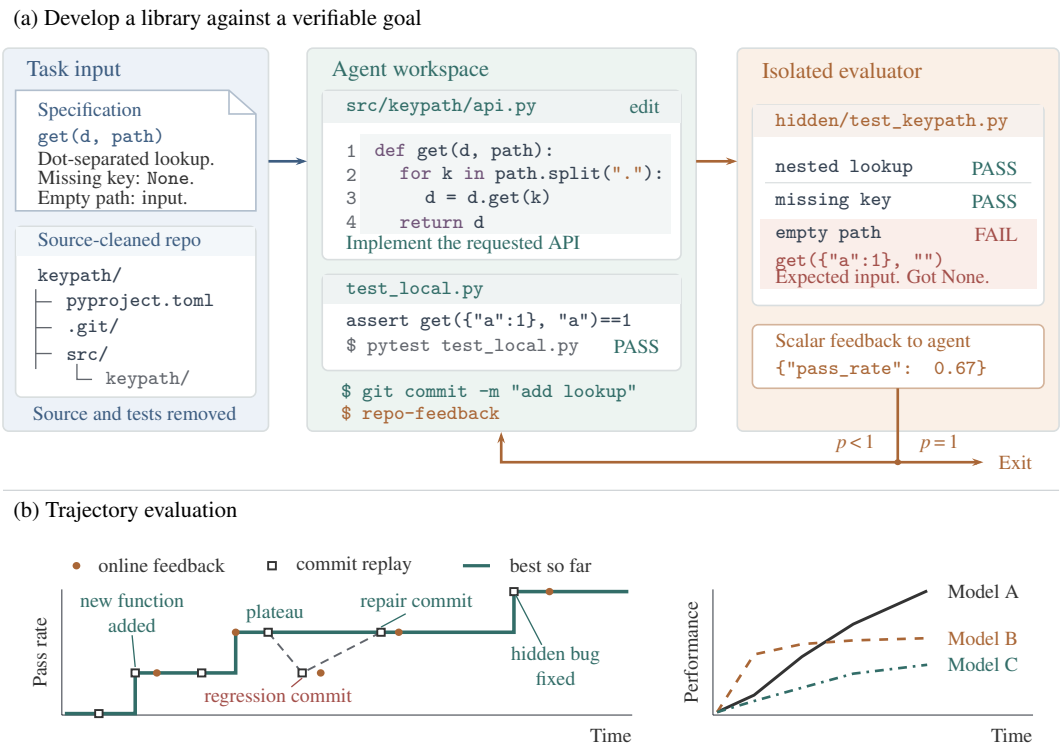


Figure 1: The UntilItPasses protocol. (a) Agents develop a library from a specification and cleaned worktree, run self-tests, and query scalar feedback from isolated hidden tests. (b) Online feedback and independent commit replay measure task progress and support model comparison over time. The code, tests, outcomes, and curves are illustrative.

development. We score these commits independently and combine the scores with feedback-call histories to compare progress across agent systems.

Our experiments show rapid early progress and difficulty reaching full-suite success. Trajectory annotations associate early gains with specification-driven implementation and later gains with feedback-guided revision. Selected cases show agents broadening self-tests to check new functionality, then revising assumptions those tests left unchallenged. Model efficiency rankings change with the required completion level, revealing different strengths in rapid initial implementation and sustained progress toward higher completion targets. These findings motivate developing and evaluating agents for both initial implementation quality and feedback-guided self-correction toward verified completion.

Contributions. We make three contributions:

- We introduce UntilItPasses, a benchmark and protocol for goal-directed repository generation with scalar feedback and explicit completion criteria.
- We develop a trajectory evaluation framework to measure verified progress and cost to success under a common budget.
- We characterize progress and revision behavior as coding agents work toward verified completion in repository generation.

2 Related Work

Specification-driven development. NL2Repo-Bench, RepoGenesis, and BeyondSWE's Doc2Repo evaluate completed repositories constructed from natural-language specifications (Ding et al., 2025; Peng et al., 2026; Chen et al., 2026). In these benchmarks, agents complete each task in a single

generation run, with deployment and black-box testing applied afterward. UntilItPasses brings feedback into the development loop, letting agents repeatedly check and revise their implementations until verified completion or budget exhaustion. SWE-Marathon covers ports, reproductions, and product clones with visible development checks and hidden final verification (Desai et al., 2026). Our repository-generation setup integrates hidden-suite feedback into repeated cycles of implementation and revision.

Minimal feedback loops. Commit0 exposes the full evaluation suite and detailed execution feedback (Zhao et al., 2024), allowing agents to tailor code to the scored inputs, expected outputs, and failure traces. UntilItPasses limits test-information leakage by withholding cases and diagnostics, returning only an aggregate pass fraction, leaving expected outputs and error diagnosis to the agent's own tests. SWE-AGI combines public tests, repeated private-suite submissions, and unrestricted web search for MoonBit system construction (Zhang et al., 2026). Our protocol keeps all official tests hidden and blocks external network retrieval, while permitting agent-authored local tests. MirrorCode and ProgramBench also block internet access but expose reference executables for behavioral reconstruction (Adamczewski et al., 2026; Yang et al., 2026). Our agents work from a written specification with no queryable reference implementation. EdgeBench returns task-defined scores, verdicts, or diagnostics across domains (Zhu et al., 2026). We limit external feedback to the hidden-suite pass fraction to study sustained revision without test-level diagnostics.

3 UNTILITPASSES

UntilItPasses evaluates repository generation through repeated development, scalar feedback, and independent trajectory measurement. We describe the tasks and benchmark features (Section 3.1) and the evaluation workflow (Section 3.2).

3.1 TASK FORMULATION

We reuse specifications, test suites, and task images from DeNovoSWE (Zhao et al., 2026) for 46 Python repository-generation tasks. The workspace is cleaned by removing implementations, tests, cached copies of the target package, and prior Git history. Build metadata and installed dependencies are preserved, so the agent can run code in an offline environment.

Each task provides a natural-language API-and-behavior specification and a Git skeleton (Figure 1(a)). The agent must produce a working library that passes a fixed hidden test suite. We measure progress by the fraction of this suite that passes.

Specifications have a median length of 23 735 tokens, and hidden suites contain 22–681 test cases per task. Selected DeNovoSWE difficulty scores range from 0.753 to 0.977 on a $[0, 1]$ scale, with higher values indicating harder tasks.

DeNovoSWE constructs specifications from source inspection and test-execution traces, with rule-based component-coverage checks and iterative critic–repair. The documentation-writing agent's prompt explicitly requires test-relevant signatures, exceptions, return fields, constants, and defaults to be stated. Full task details, selection and validation procedures, and the task list are provided in Appendix A.3.

3.2 EVALUATION WORKFLOW

We run tasks with Harbor (Harbor Framework Team, 2026), allowing agents to edit code, run self-authored tests, and query the hidden-suite pass rate of their current workspace at any point (Figure 1(a)). Agents choose when to request feedback and how to revise their implementations, committing changes throughout development. A verified perfect score or budget exhaustion ends development. If an agent stops earlier, the harness resumes the same conversation and workspace as long as budget remains. Initialization and continuation prompts and context window sizes are provided in Appendix A.

We independently evaluate the committed history after the agent and its subprocesses have stopped. Each within-budget commit in the final Git history is scored in an isolated evaluator container with no network access. The evaluator uses the same fixed test suite and 120-second timeout as online

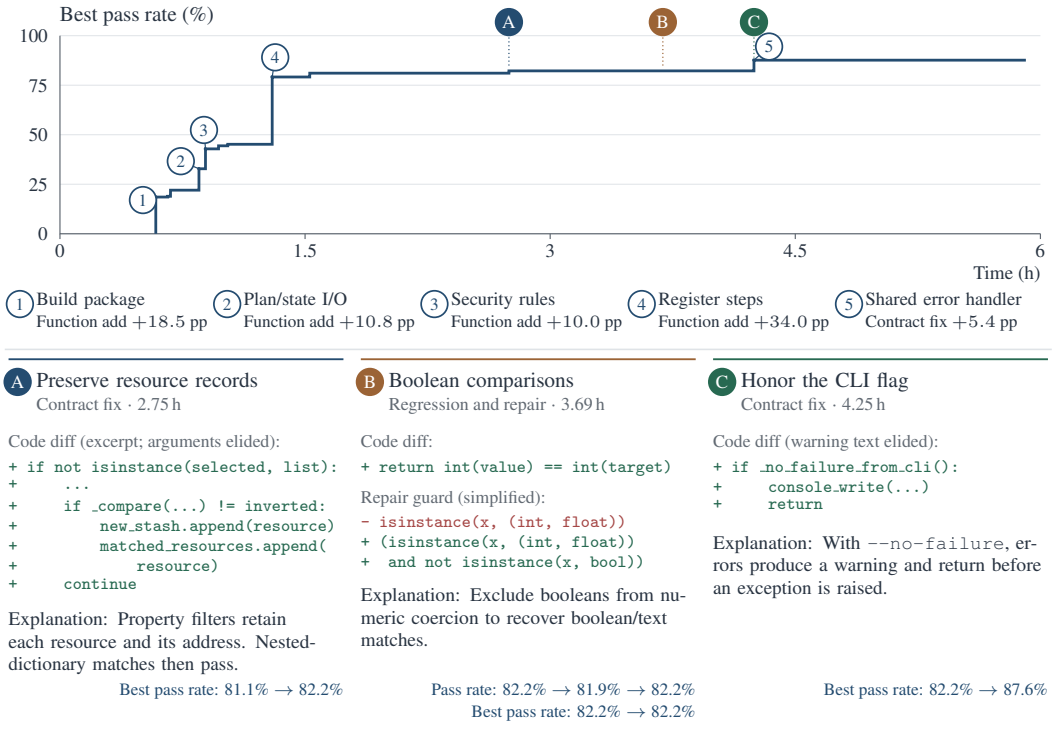


Figure 2: Best-pass-rate progress over time of Hy4 preview on `terraform-compliance`. Numbered markers identify the five largest lifts. A/B/C mark selected repair cases shown in the code diffs and explanations: a filter repair that raises the best score (A), a boolean-comparison regression recognized and repaired with the best pass rate unchanged (B), and CLI-flag handling extended to direct error-handler calls (C).

feedback. Each score is placed at the budget point where the corresponding commit was produced. Combining these scores with online feedback yields a *best-pass-rate* trajectory over the consumed budget. The best pass rate at any point is the maximum valid score observed so far (Figure 1(b)). Figure 2 shows a measured trajectory, with additional task trajectories in Appendix C.

Development and grading run in separate containers: hidden tests, fixtures, and scoring logs remain in the evaluator, and feedback exposes only the pass fraction. Agent network access is restricted to model API endpoints, preventing retrieval of upstream source code.

UntilItPasses has three features:

- **Goal-directed.** Passing every hidden test defines the delivery criterion. The agent works toward verified completion within a fixed budget, testing whether it can iteratively build and correct a library toward a concrete target.

- **Minimal feedback.** Every task exposes the same aggregate-only feedback channel. The agent diagnoses failures through specification reading, code inspection, and self-authored tests, revealing how the agent exploits a sparse correctness signal.

- **Measurable development progress.** Scores along the development history reveal when functionality is established, where progress stalls, and whether later revisions improve completion. A common budget scale supports comparisons of both achieved quality and work required.

4 EXPERIMENTS

Section 4.1 states the conditions and resources, Section 4.2 defines the evaluation metrics, and Section 4.3 reports outcomes over the 552 trials. Sections 4.4 and 4.5 assess seed and speed-calibration sensitivity, and Section 4.6 examines single-pass and feedback-loop evaluation.

4.1 EXPERIMENTAL SETTINGS

The evaluation covers twelve agent systems on the same 46 tasks, giving 552 comparable trials. Each condition uses its provider's highest thinking-effort tier and either its native scaffold or Claude Code. Tool settings and context window sizes appear in Appendix A.4.

Each candidate receives one CPU, 4,096 MB memory, and 20,480 MB storage. The agent can execute and test code locally, query the live evaluator through `repo-feedback`, and receive a pass rate computed with a fixed task-specific test count (Section 3.2).

We convert a common reference duration into a token-equivalent budget using each model's pre-launch generation speed. Using token budget could reduce the impact of serving and API interruptions in long-horizon agent runs documented by EdgeBench (Zhu et al., 2026). Speeds are primarily from OpenRouter's averaged output throughput over a dated seven-day window. A session interrupted by an infrastructure failure is resumed, retaining its prior usage and the whole-trial budget. Accounting, stopping rules, and each model's calibration dates are detailed in Appendix B.

Table 1: Leaderboard by weighted reach-rate AUC A, with each model's scaffold and reasoning effort. Token budgets and speed calibration appear in Table 4.

Model	Scaffold	Effort	A
Opus 5	Claude Code	max	0.726
Grok 4.6	Grok Build	xhigh	0.628
GPT-5.6 Sol	Codex	max	0.512
DSV4.1 Flash	DSH	max	0.485
Kimi K3	Kimi Code	max	0.472
Hy4 preview	Claude Code	high	0.467
GLM-5.3	ZCode	max	0.457
Gemini 3.7 Flash	Claude Code	high	0.420
Sonnet 5	Claude Code	max	0.392
DSV4 Pro 0813	DSH	max	0.372
Qwen 3.8 Max	Claude Code	xhigh	0.363
DSV4 Flash 0731	DSH	max	0.311

4.2 EVALUATION METRICS

The best pass rate (Section 3.2) summarizes progress observed through online feedback and post-run commit evaluation within the budget.

To reward greater completion, our main aggregate metric, *weighted reach-rate AUC*, weights each percentage threshold T by its value. For task i, let $p_i \in [0, 1]$ be its best pass rate and define $r(T) = \text{mean}_i\ \mathbf{1}[100p_i \geq T]$, the share of tasks reaching $T\%$. Then

$$A = \frac{\sum_T T\, r(T)}{\sum_T T}, \qquad T \in \{5, 10, \ldots, 100\}. \tag{1}$$

The 5-percentage-point grid reduces sensitivity to fine-grained noise from uneven feature coverage, with equal task weights regardless of suite size. Restricting observations to budget fraction x gives $A(x)$, measuring progress throughout a run. Mean squared best pass rate is the continuous counterpart. Its derivation and discretization comparison are in Appendix D.1.

4.3 MAIN RESULTS

Leading systems improve at different stages. Table 1 reports each system's weighted reach-rate AUC. Figure 3 shows the share of tasks reaching each pass-rate threshold and how AUC changes as the budget is consumed, spanning a range of 2.34×. Opus 5 has the highest reported AUC at 0.726, followed by Grok 4.6 at 0.628 and GPT-5.6 Sol at 0.512. DSV4.1 Flash, Kimi K3, Hy4 preview, and GLM-5.3 have closely spaced scores. Grok 4.6 reaches 85.2% of its final A by 25% of the budget. At the same point, Opus 5 has reached 22.8% of its final A, but it overtakes the other leaders at 45% and leads thereafter. After 50% of the budget, Opus 5 adds 0.113 to A, compared with 0.057 for Grok 4.6 and 0.024 for GPT-5.6 Sol.

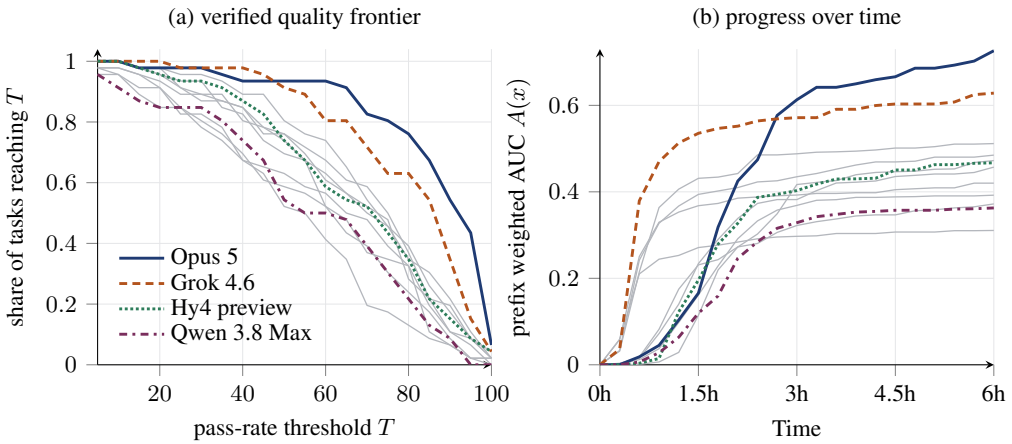


Figure 3: Task completion and progress over time. (a) Share of 46 tasks whose best pass rate reaches each threshold T. Thresholds use the same 5-percentage-point sampling grid as weighted AUC (Eq. (1)). (b) Weighted AUC $A(x)$ over time, with x the fraction of the six-hour allowance consumed. Both panels show Opus 5, Grok 4.6, Hy4 preview, and Qwen 3.8 Max in color. Table 1 gives every condition's A.

Trials establish a partial implementation early. Most trials reach a nonzero best pass rate quickly, with the first such gain arriving at a median 7.5% of the budget. The trajectories then diverge as systems address the remaining implementation problems.

Later progress shifts toward revision. Further gains involve sustained revisions as agents repair behavior and revisit assumptions left unchallenged by initial self-tests.

4.4 Seed sensitivity

Due to cost constraints, we limit the multi-seed experiment to DSV4.1 Flash, repeating it ten times on the same 46 tasks while holding the harness, reasoning tier, compaction, six-hour horizon, token budget, and feedback channel fixed. Mean weighted reach-rate AUC is 0.4759 (SD: 0.0215, 95% Student-t CI: [0.4605, 0.4913]). Best pass rates vary on every task, and solved task sets differ across runs. Per-run results and task-level variability are detailed in Appendix D.2.

4.5 Speed calibration sensitivity

We assess sensitivity to downward speed-calibration errors by evaluating each system's AUC on a prefix of its recorded trajectory, conservatively bounding the effect of reducing generation speed by up to 10%. Allowing systems to vary independently preserves 64 of 66 pairwise orders. The overlapping pairs involve Kimi K3 with Hy4 preview and with GLM-5.3. These bounds fix the recorded trajectories and accounting model; wall-clock costs and behavioral responses to a changed budget remain outside this analysis. The envelope derivation, each system's interval, and rank bounds are detailed in Appendix B.3.

4.6 Single-Pass and Feedback-Loop Evaluation

To illustrate the motivation for our feedback-loop evaluation setup, we provide a single-pass comparison in which the agent stops at its first voluntary completion. We use DSV4.1 Flash with the same tasks, harness, reasoning tier, and compaction settings, without a scoring tool, forced continuation, or preset token-equivalent budget. Its final workspace yields a weighted AUC of 0.4004. Elapsed agent wall time averages 42.74 minutes (median 40.36, interquartile range 29.88–46.34), excluding setup and evaluation.

In Prometheus, the single-pass final workspace omits the required `_total` suffix for base counter samples. The separate feedback-loop run initially makes the same error, then revisits counter naming

and retains a correction after improved feedback. The comparison concerns the full protocols, which differ in feedback, continuation, work allowance, and score selection. The specification, paired code, and recorded feedback are detailed in Appendix E.6.

5 Analysis and Discussion

Recorded trajectories show early broad implementation followed by testing and revision. Stronger systems gain an advantage both in the properties they implement correctly from the start and in the obstacles they later resolve. We examine this progression through lift timing, budget efficiency across completion targets, and paired code properties.

5.1 Early implementation and late repair

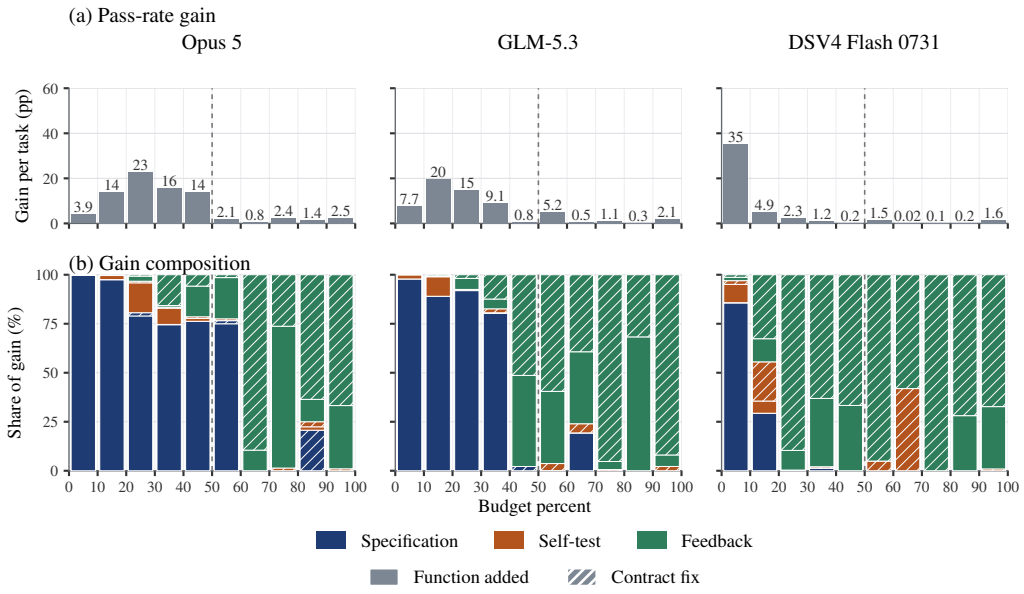


Figure 4: Lift attribution for Opus 5, GLM-5.3, and DSV4 Flash 0731, spanning strong, middle, and weak performance across distinct model families. Each column shows one model in 10%-wide budget intervals. The upper row sums pass-rate gains from function additions and contract fixes, divided by all 46 tasks per model, on a common scale. The lower row shows each bin's gain composition. Colors identify the information source and hatching marks contract fixes. Other change types are excluded. The dashed line marks half the budget. Lift types and information sources use exploratory LLM-assisted annotations. All-model plots, annotation methods, coverage, and validation limits are in Appendix C.1.

Trajectory annotations suggest a shift from specification-driven implementation early in the budget to feedback-guided revision later. We define a *lift* as a scored checkpoint whose pass rate exceeds every previous scored checkpoint in the trial, starting from an initial pass rate of zero. Lift occurrence, magnitude, and budget position come from recorded test results and execution metadata. LLM-assisted annotations characterize the associated changes and their apparent information sources: the specification, self-testing, or scalar feedback. We focus on adding functionality and repairing an implementation to satisfy the specification. The annotations cover 5,481 lifts across 12 systems. Figure 4 shows representative systems.

Early gains are mainly labeled as function additions. Under our annotation scheme, function-adding changes before the 50% budget mark account for 78.9–97.4% of the pass-rate improvement across systems in annotated trajectories. Most of these gains are attributed to the specification. These shares fall to 9.3–50.7% after the mark.

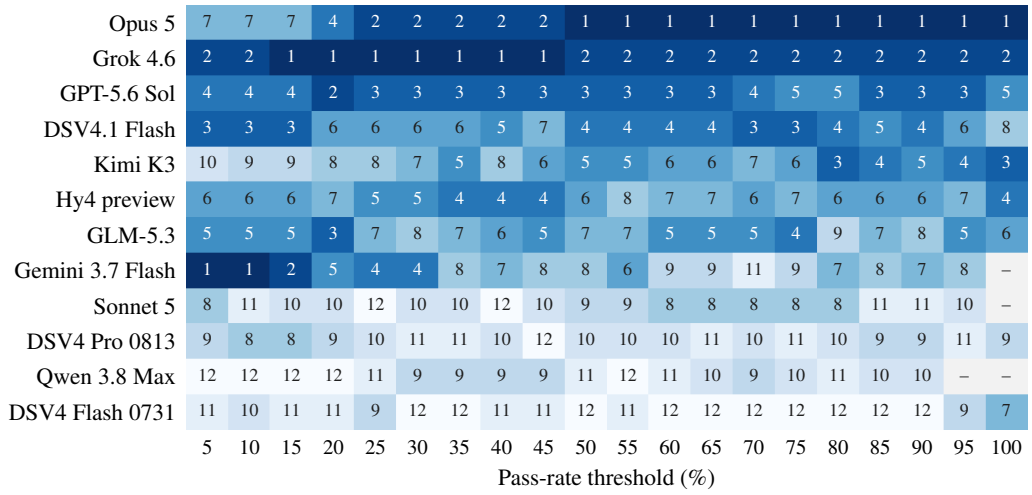


Figure 5: The leading model changes with the required completion level. Ranks use the same 46 tasks, ordered by reached-task count, then mean first-crossing budget time. Darker cells indicate higher ranks. A dash denotes no reached task. Rows follow the main leaderboard order.

Feedback-guided revision in later gains. Changes labeled as feedback-guided account for most of the plotted second-half pass-rate gain. These include added functionality and contract repairs, with repairs taking a growing share of later gains. Feedback-guided labels identify recorded links between scalar readings and subsequent investigations or revisions. These category shares describe observed gains conditional on the assigned labels.

Table 2: Weighted reach-rate AUC credit from gains by half budget (Early) and after it (Late), over all 552 trials. Early and late sum to A before rounding. Rows follow the evaluation-table order.

Model	A	Early	Late
Opus 5	0.726	0.613	0.113
Grok 4.6	0.628	0.572	0.057
GPT-5.6 Sol	0.512	0.488	0.024
DSV4.1 Flash	0.485	0.435	0.050
Kimi K3	0.472	0.417	0.055
Hy4 preview	0.467	0.403	0.064
GLM-5.3	0.457	0.382	0.075
Gemini 3.7 Flash	0.420	0.393	0.028
Sonnet 5	0.392	0.369	0.024
DSV4 Pro 0813	0.372	0.322	0.050
Qwen 3.8 Max	0.363	0.329	0.034
DSV4 Flash 0731	0.311	0.297	0.014

Observed roles of self-tests and feedback. The continuation policy keeps unfinished trials active. Within these trials, selected trajectories show agents using scalar readings to formulate hypotheses, broaden self-tests, and revise implementation assumptions. During early broad implementation, self-tests help check whether intended functionality has been implemented. Code and self-tests can share the same mistaken assumption, so local tests may already pass before a repair. Correcting that assumption enables both the implementation and its tests to capture the intended behavior. The cases in Section 5.3 illustrate how feedback shapes what agents investigate and which implementation assumptions they revisit.

5.2 Budget efficiency across completion targets

To compare model efficiency at different levels of task difficulty, we vary the required completion level on the same tasks. Weighted reach-rate AUC rewards progress toward full completion. Fixing a pass-rate target τ relaxes the success criterion and truncates the evaluation objective at τ. Each threshold ranks models by **reached-task count, descending**, then **mean first-crossing budget time, ascending**, over reached tasks. The same recorded tasks and budgets underlie every threshold. Time conversion and complete tables are detailed in Appendix B.4.

Low completion targets favor Gemini's fast initial progress. Gemini 3.7 Flash leads below 15%, while Opus 5 ranks seventh (Figure 5). At 10%, both reach all 46 tasks, taking 19.4 and 88.3 budget

minutes on average. Gemini therefore achieves the same coverage with substantially less budget when the completion requirement is low.

Higher completion targets favor frontier models. Grok 4.6 leads at 15%–45%, and Opus 5 at 50%–100%. At 80%, Opus 5 reaches 35 tasks and Gemini 3.7 Flash reaches 15. As the completion requirement rises, the advantage shifts toward models that sustain progress across more tasks. This pattern is consistent with the feedback-guided late improvement in Section 5.1. Section 5.3 traces how feedback supports successive revisions.

5.3 Case Study

The cases illustrate how feedback shapes investigation (Section 5.1), with agents differing in the tests and assumptions they revisit. In Cleo, repeated flat feedback prompts Sonnet 5 to broaden self-tests and repair command parsing. A gain then prompts a related internal-call hypothesis. Subsequent internal-call and help repairs pass local checks while feedback remains unchanged. Opus 5 repairs the same public-input property earlier through an application test. In Defcon, Opus uses feedback comparisons to focus on external-change detection, then separates disk snapshots from unsaved local edits, producing a verified gain. GLM-5.3 also probes detection, retains its assumption that locally edited files can be skipped, and shifts to notification handling. Task explanations, agent reasoning and actions, model comparisons, and verification are detailed in Appendices E.1–E.5.

6 Conclusion and Limitations

Conclusion. UntilItPasses evaluates whether coding agents sustain progress toward a verifiable goal in repository generation. Its trajectory measurements characterize progress within a fixed budget, while behavioral analyses document how agents investigate and revise incomplete implementations. Exploratory trajectory annotations suggest a shift from broad functionality development toward repair. Changes labeled as feedback-guided account for most of the plotted later gain. Paired cases illustrate how agents revise assumptions shared by their implementations and self-tests. Model rankings also change with the required completion level, revealing different advantages in fast initial progress and in sustained progress toward higher completion targets. These findings highlight feedback-guided self-correction as a key capability for autonomous coding agents. UntilItPasses provides a testbed for developing and evaluating this capability alongside initial implementation quality and measuring the budget required to reach verified completion.

Limitations. Tasks derive from public repositories, so pretraining may already contain an upstream implementation. Calibrated token budgets reduce sensitivity to model-speed fluctuations and infrastructure interruptions, but may not fully reflect real-world usage time. We have not systematically studied how the number of feedback calls affects progress. We prompt agents to commit after each API change, and these prompted commits provide only a coarse view of progress. Behavioral annotations lack systematic validation against independent human labels, leaving category shares conditional on the annotation procedure. Benchmark size is bounded by compute cost, limiting the number of tasks and repeated trials.

Use of AI Assistance

We used large language models to improve manuscript clarity, assist with LaTeX formatting, and polish writing. We used DSV4.1 Flash for exploratory annotation of pass-rate lifts and their information sources, including a separate LLM pass for feedback attribution. The annotation procedure, targeted evidence checks, and validation limits are described in Appendix C.1. The authors take full responsibility for the research design, analyses, conclusions, and final content of this paper.

Reproducibility Statement

Code, data, and trajectories will be made publicly available upon acceptance. The evaluation workflow and experimental settings are described in Sections 3.2 and 4.1. Selection criteria, identifiers for all 46 tasks, prompts, and model configurations appear in Appendix A, with budget accounting and speed calibration in Appendix B.

References

Tom Adamczewski, David Owen, David Rein, Florian Brand, Giles Edkins, Allen Hart, and Daniel O'Connell. MirrorCode: AI can rebuild entire programs from behavior alone. *arXiv preprint arXiv:2606.30182*, 2026.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, Ruihua Song, Chang Liu, Cheng Chen, Kai Jia, and Ji-Rong Wen. BeyondSWE: Can current code agent survive beyond single-repo bug fixing? *arXiv preprint arXiv:2603.03194*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Daniel Wang, Derek Chen, Pranav Raja, Albert Liu, Marek Šuppa, Nevasini Sasikumar, Luyang Kong, Erik Quintanilla, Xiangyi Li, Ivan Bercovich, and Steven Dillmann. SWE-Marathon: Can agents autonomously complete ultra-long-horizon software work? *arXiv preprint arXiv:2606.07682*, 2026.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. https://github.com/harbor-framework/harbor, January 2026.

Zhiyuan Peng, Xin Yin, Pu Zhao, Fangkai Yang, Lu Wang, Ran Jia, Xu Chen, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. RepoGenesis: Benchmarking end-to-end microservice generation from readme to repository. *arXiv preprint arXiv:2601.13943*, 2026.

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

Zhirui Zhang, Hongbo Zhang, Haoxiang Fei, Zhiyuan Bao, Yubin Chen, Zhengyu Lei, Ziyue Liu, Yixuan Sun, Mingkun Xiao, Zihang Ye, Yu Zhang, Hongcheng Zhu, Yuxiang Wen, and Heung-Yeung Shum. SWE-AGI: Benchmarking specification-driven software construction with MoonBit in the era of autonomous agents. *arXiv preprint arXiv:2602.09447*, 2026.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. DeNovoSWE: Scaling long-horizon environments for generating entire repositories from scratch. *arXiv preprint arXiv:2606.10728*, 2026.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024. URL https://arxiv.org/abs/2412.01769.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, Yueyang Wang, Junfeng Zhong, Rui Liu, Tian Gao, Yangguang Lin, Jingyuan Zhang, Maojia Song, Xuan Qi, Jinhong Wu, Chenyang Zhang, Yinzhu Piao, Ziru Niu, Hongbin Lin, Lingxiang Meng, Peng Tang, Chengyao Tang, Shanyu Wu, Huanyu Zheng, Yu Liu, Liya Zhu, He Wang, Ming Ding, Ziyu Wan, Hao Liu, Sibo Wang, Haotian Zhu, Xintian Zhang, Nan Chai, Yipeng Liu, Panhao Lai, Sihang Yuan, Zixin Su, Ge Zhang, Wangchunshu Zhou, Yantao Du, Wenhao Huang, and Guang Shi. EdgeBench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026.

A Prompts, task details, and context windows

A.1 Initial prompt

Every task's instruction opens with the shared preamble below and continues with that task's repository specification. This preamble is the complete human input for a trial apart from the specification itself: it states the workspace, the development discipline, the feedback command, and the terminal condition.

````
You are in `/workspace`. Implement the Python repository described by the
specification below and pass all hidden unit tests. The required dependencies
are already installed. The network is disabled. Git user name and email are
already configured for the repository agent.

Before coding, read the full specification and map the dependencies between its
APIs. Implement in dependency order: start with shared constants, exceptions,
data models, and low-level helpers; then build the core state and behavior that
depends on them; finish with higher-level public APIs, integrations, and entry
points. At each step, choose an API whose direct dependencies are already
implemented, and keep the package importable and internally consistent.

Implement one API change at a time. After implementing or modifying any public
or private function, method, class, entry point, loader, parser, wrapper, or
helper, commit it before starting another API change. Keep the worktree clean
between API changes. Never run `git reset` in this workspace. Preserve the
existing workspace and commit history, and make every change incrementally on
top of the work already completed.

You can obtain the hidden-unit-test pass rate by running a feedback loop when it
is useful. Make sure the worktree is clean, then run:

```bash
repo-feedback
```

This tool tests the committed repository snapshot and prints its hidden-unit-test
pass rate, for example:

```json
{"pass_rate":0.75}
```

Obtain feedback only by running the exact `repo-feedback` command shown above.
Do not inspect or modify the feedback tool, its configuration, environment, or
implementation. Do not probe the feedback service, contact its endpoints
directly, construct or send custom requests or snapshots, change its timeout,
or try to obtain feedback by any other means.

Use numeric pass rates to guide further implementation. Continue developing the
repository until all hidden tests pass. A `repo-feedback` evaluation reaching
`{"pass_rate":1.0}` is terminal: the harness ends the agent run before the
command returns control. No command suffix, following shell action, or later
model turn runs after that result.

Run each feedback call in the foreground and wait for its final JSON before
starting another feedback call.

Each `repo-feedback` call has a fixed 120-second timeout. If a call times out,
it exits with status `0` and prints
````

```
`{"pass_rate":null}`. This means no valid score was produced; it is not a zero,
partial, or stale score, and the tool does not retry automatically. A `null` can
also mean that the snapshot, request, service, or evaluator failed. Check that
the worktree is clean and no earlier call is running, address the likely issue,
and try again.

A timeout or `null` leaves the agent run active, so continue developing until a
fresh feedback evaluation reaches numeric `1.0`.
```

A.2 CONTINUATION PROMPT

A voluntary exit with code zero leaves the episode active while budget remains. The managed lifecycle then resumes the same conversation and workspace with the text below, which carries no task content and no feedback call.

```
Continue working on the task. Review the current workspace and the work
already completed, make further progress, and use repo-feedback when useful.
Do not stop while the task remains incomplete.
```

Each occurrence of this text in a transcript marks one resume boundary, so the count of continuations and the stopped agent time between them are recoverable from the transcript alone. Agent runtimes that resume through their own native session mechanism record their own resume marker in place of this text.

A.3 RELEASED TASK SET

DeNovoSWE (Zhao et al., 2026) supplies a specification, repository revision, executable environment, hidden suite, and fixtures for each task. Specification length is measured in `o200k_base` tokens, excluding shared harness instructions. Lengths range from 10 801 to 35 341, with a median of 23 735; hidden suites contain 22–681 test cases per task. We review reference-implementation runs and task defects, excluding environment and evaluator-contract failures that block installation or test collection. The selected tasks have the following properties:

- **Difficulty.** Selected tasks span source difficulty scores from 0.753 to 0.977. DeNovoSWE combines normalized executable Python line counts with two LLM ratings of the specification, calibrating the weights against DeepSeek-V4-Pro's mean rollout pass rate. Scores lie in $[0, 1]$, with higher values indicating harder tasks.

- **Reference compatibility.** Recorded reference-implementation runs pass the selected suite.

- **Evaluability.** The suite completes on the reference implementation within the 120-second evaluation limit.

- **Environment health.** Recorded runs show no task-environment failure requiring exclusion.

These checks establish executability and reference compatibility.

The 46 selected tasks appear below by identifier, each naming its upstream repository and pull request. The list is sorted.

astropy_pyvo_pr323
authlib_joserfc_pr63
bdaiinstitute_spatialmath-python_pr166
c2nes_javalang_pr70
cognitedata_expression_pr224
datafolklabs_cement_pr647
dwavesystems_dwave-cloud-client_pr479
ebellocchia_bip_utils_pr18
ebroecker_canmatrix_pr543
gitguardian_ggshield_pr493
google_langextract_pr261
graphql-python_gql_pr328
growthbook_growthbook-python_pr77
hdwallet-io_python-hdwallet_pr73
indigo-dc_udocker_pr389
iterative_dvc-data_pr622
jtesta_ssh-audit_pr252
karask_python-bitcoin-utils_pr166
kelsin_mysql-mimic_pr69
keon_algorithms_pr868
kludex_starlette_pr672
linode_linode_api4-python_pr290
lyft_confidant_pr444
maecproject_python-maec_pr90
matan1008_pykdebugparser_pr20
mathandy_svgpathtools_pr238
microsoft_fabric-cicd_pr632
miguelgrinberg_python-socketio_pr1241
ncclient_ncclient_pr645
nylas_nylas-python_pr444
oasis-open_cti-pattern-validator_pr81
oauthlib_oauthlib_pr910
openfga_python-sdk_pr75
petl-developers_petl_pr463
prometheus_client_python_pr1139
pyasn1_pyasn1-modules_pr12
python-poetry_cleo_pr512
qua-platform_py-qua-tools_pr196
redimp_otterwiki_pr282
robotools_defcon_pr469
rokam_midea-local_pr412
seequent_properties_pr290
spotify_chartify_pr174
terraform-compliance_cli_pr762
us-irs_spacepackets-py_pr127
wireservice_agate_pr785

A.4 CONTEXT WINDOW SIZES

Table 3 lists each model's scaffold, thinking-effort tier, context capacity, maximum output, and the context size at which compaction starts, using the settings recorded for each trial. The compaction threshold is the context capacity less the maximum output. GPT-5.6 Sol uses its provider's recommended compaction threshold of 272,000 tokens. Model speeds and token ceilings appear in Table 4.

DeepSeek Harness (DSH) uses the minimal preset with `bash` and `str_replace_editor`, with native automatic compaction enabled.

Every condition runs at one explicit thinking-effort tier, the highest the model provider supports: max on Opus 5, Sonnet 5, GPT-5.6 Sol, Kimi K3, GLM-5.3, and both DSV4 Pro 0813 and DSV4 Flash 0731. The tier is xhigh on Grok 4.6 and Qwen 3.8 Max, and high on Gemini 3.7 Flash and Hy4 preview. Each tier name denotes the highest setting offered by its provider. The selected tier is part of each system's configuration. Table 1 reports the same tier for each system in the main text.

Table 3: Per-model scaffold, thinking-effort tier, context capacity, compaction threshold, and maximum output. Context and output sizes are in tokens. Models follow the order of Table 1.

Model	Scaffold	Effort	Context window	Compaction threshold	Maximum output
Opus 5	Claude Code	max	1,000,000	872,000	128,000
Grok 4.6	Grok Build	xhigh	500,000	300,000	200,000
GPT-5.6 Sol	Codex	max	1,050,000	272,000	128,000
DSV4.1 Flash	DSH (minimal + compact)	max	1,000,000	616,000	384,000
Kimi K3	Kimi Code	max	1,048,576	792,576	256,000
Hy4 preview	Claude Code	high	1,048,576	786,432	262,144
GLM-5.3	ZCode	max	1,000,000	872,000	128,000
Gemini 3.7 Flash	Claude Code	high	1,048,576	1,016,576	32,000
Sonnet 5	Claude Code	max	1,000,000	936,000	64,000
DSV4 Pro 0813	DSH (minimal + compact)	max	1,048,576	664,576	384,000
Qwen 3.8 Max	Claude Code	xhigh	1,000,000	968,000	32,000
DSV4 Flash 0731	DSH (minimal + compact)	max	1,000,000	616,000	384,000

B BUDGETS AND CALIBRATION

B.1 BUDGET ACCOUNTING AND STOPPING CRITERIA

A trial's token-equivalent ceiling is $B = \lceil sd \rceil$, where s is its pre-launch generation speed and d a common reference duration. Let u denote completed-response output tokens from the root session and w the duration of the union of completed ordinary-tool intervals, feedback calls included, and calibrated intervals spent by subagents spawned by that session (descendant work). Effective consumed budget is

$$e = u + s \cdot w. \tag{2}$$

Subagent output converts to duration at the same speed, and the union charges overlapping tools and branches once along their critical path, so generation, tool use, and descendant work share one coordinate. Lifecycle hooks enforce the first observed crossing of B, and the frozen native transcript reproduces the accounting after the run.

Feedback calls run in the foreground under a 120-second limit, and a null, timed-out, or errored call yields no score and leaves the trial active. A trial interrupted by infrastructure failure, missing transcript data, or failure to stop all agent processes must be resumed with its prior resource use and original budget preserved before it can be included in the evaluation. During long runs, native runtimes compact conversations into summaries and continue from retained context, so compaction events consume active work and can affect subsequent development.

B.2 GENERATION-SPEED CALIBRATION

All trials use a six-hour reference duration, $d = 21{,}600$ seconds, converted to a token ceiling using each model's generation speed. Table 4 lists each system's generation speed s, its integer token-equivalent ceiling B, and source. Public-series entries average the selected endpoint's available daily median output throughput over a seven-day UTC window. Accounting details are in Appendix B.1.

Table 4: Reference generation speeds and token budgets, one fixed speed and token ceiling per model. Speeds are output tokens per second. Ceilings are token-equivalent units. Dates are in 2026. Public-series windows use UTC and include both endpoints. Public speeds average daily medians over seven days.

Model	s	B	Source and observation dates
Opus 5	62.67	1,353,672	OpenRouter: Opus 5, Aug. 7–13
Grok 4.6	56.50	1,220,400	OpenRouter: Grok 4.6, Aug. 7–13
GPT-5.6 Sol	36.43	786,888	OpenRouter: GPT-5.6 Sol, Aug. 6–12
DSV4.1 Flash	87.00	1,879,200	Local DSH measurement, adopted Sept. 9, measurement date unrecorded
Kimi K3	26.71	576,936	OpenRouter: Kimi K3, Aug. 6–12
Hy4 preview	30.00	648,000	Local agent measurement; tested Aug. 29
GLM-5.3	48.00	1,036,800	OpenRouter: GLM-5.2 proxy, July 23–29
Gemini 3.7 Flash	78.71	1,700,136	OpenRouter: Gemini 3.7 Flash, Aug. 19–25
Sonnet 5	53.29	1,151,064	OpenRouter: Sonnet 5, Aug. 17–23
DSV4 Pro 0813	43.64	942,624	OpenRouter: DSV4 Pro proxy, July 23–29
Qwen 3.8 Max	40.17	867,672	OpenRouter: Qwen 3.8 Max, Aug. 7–13
DSV4 Flash 0731	84.00	1,814,400	OpenRouter: DSV4 Flash, Aug. 7–13

Public-series dates identify completed UTC days. The stored end timestamp is midnight after the last listed day. We average the available daily medians at the default service tier, preferring official providers and otherwise the highest available precision. Provider, endpoint, quantization, and tier are pinned in the calibration record. Two conditions use related model series: GLM-5.3 uses GLM-5.2 and DSV4 Pro 0813 uses DSV4 Pro, both over July 23–29, 2026.

We tested Hy4 preview locally on Aug. 29, 2026, and used a conservative 30.00 output tokens/s. DSV4.1 Flash uses a retained DSH measurement of 87.00 output tokens/s, adopted on Sept. 9, 2026. Its original measurement date was not recorded. Both calibrations retain predecessor OpenRouter fields as provenance metadata, separate from their measurement dates.

B.3 SPEED CALIBRATION SENSITIVITY

For each selected trial, let B_i be its recorded token-equivalent ceiling and $e_i(T)$ the budget consumed when the best pass rate first reaches threshold T. AUC at budget fraction x uses the same tasks, scores, thresholds, and weights as the main metric:

$$A_m(x) = \frac{\sum_{T\in\mathcal{T}} T\left(\frac{1}{N}\sum_{i=1}^{N}\mathbf{1}\{e_i(T)\le xB_i\}\right)}{\sum_{T\in\mathcal{T}} T}, \qquad \mathcal{T}=\{5,10,\dots,100\}. \tag{3}$$

An unreached threshold contributes zero. Each system uses the same 46 tasks and their recorded token ceilings. Monotonicity gives the interval $[A_m(\rho), A_m(1)]$ for every independently chosen budget fraction $x_m \in [\rho, 1]$, with $\rho = 0.9$. The lower endpoint corresponds to a 5.4-hour prefix of the six-hour effective horizon.

Interpretation as a speed-error bound. Hold the recorded output, ordinary-tool durations, observation attribution, and dependency graph fixed, and replace each system's calibrated speed s by $s' = \alpha s$, where $\alpha \in [\rho, 1]$ may differ across systems. Every root or descendant generation duration scales by $1/\alpha$, while ordinary-tool durations stay fixed. The accounting schedule combines these nonnegative durations through sums and critical-path maxima, so its effective time satisfies

$$t(s) \le t(s') \le \frac{t(s)}{\alpha} \le \frac{t(s)}{\rho}. \tag{4}$$

Using the reference horizon B_i/s, every observation inside the ρ-budget prefix remains within budget under the perturbed speed. With $A_m^{\mathrm{speed}}(\alpha)$ denoting AUC after this fixed-trajectory recalculation, the bound is

$$A_m(\rho) \le A_m^{\mathrm{speed}}(\alpha) \le A_m(1). \tag{5}$$

The prefix lower endpoint is a conservative bound. The exact speed-adjusted value depends on the generation and tool-work composition. Descendant generation follows the same speed perturbation and preserves its recorded parallel dependencies.

Table 5: Weighted-AUC envelopes from the 90%–100% budget prefixes of recorded trajectories, ordered by main AUC. Rank bounds allow independent prefix fractions across systems. Overlapping intervals identify orders left unresolved by the speed-error bound. These are deterministic sensitivity envelopes. Values are rounded for display, and comparisons use unrounded values.

Model	$[A_m(0.9), A_m(1)]$	Rank bounds
Opus 5	[0.693, 0.726]	1
Grok 4.6	[0.608, 0.628]	2
GPT-5.6 Sol	[0.510, 0.512]	3
DSV4.1 Flash	[0.480, 0.485]	4
Kimi K3	[0.453, 0.472]	5–7
Hy4 preview	[0.463, 0.467]	5–6
GLM-5.3	[0.441, 0.457]	6–7
Gemini 3.7 Flash	[0.409, 0.420]	8
Sonnet 5	[0.391, 0.392]	9
DSV4 Pro 0813	[0.363, 0.372]	10
Qwen 3.8 Max	[0.360, 0.363]	11
DSV4 Flash 0731	[0.310, 0.311]	12

Independent perturbations and scope. A pair retains its strict order throughout these envelopes when $A_a(\rho) > A_b(1)$. The rank bounds compare each system's lower endpoint against every other upper endpoint, and its upper endpoint against every other lower endpoint. This permits model-specific calibration errors within the stated range. For comparison, evaluating every system at the

90% budget prefix reduces AUC by 1.99% on average relative to each system's main AUC, with rank correlation $\rho_{\mathrm{Spearman}} = 0.993$. The analysis fixes the observed trajectories and recorded resource categories. It leaves input processing, first-token latency, provider-specific token accounting, and changes in agent behavior untested. Upward speed corrections can admit additional work beyond the retained trajectories and require further observations.

B.4 Budget efficiency across completion targets

Efficiency combines broad task coverage with budget left for further progress. We vary completion targets on the same 12 models and 46 tasks, from initial progress to more complete implementations, using the original trajectories and budgets.

Table 6 shows this effort as *budget time* on the common six-hour reference scale and as *budget percent*, the share of the allowance consumed. With q effective tokens at first crossing, token ceiling B, and reference duration d in seconds,

$$t = \frac{d}{60}\frac{q}{B} \text{ minutes}, \qquad b = 100\frac{q}{B}\ \%.$$

Both costs normalize the same consumed-token coordinate by the same budget. Budget time expresses normalized effort on the reference scale; actual elapsed time also depends on serving and interruptions. Effective tokens cover root output and active work. Crossings use the earliest admitted score or feedback reaching the target, with feedback placed at its attributed commit.

Models rank by reached-task count, then mean budget time. Both cost columns average only reached tasks, so their values should be read alongside coverage. Unreached tasks remain in the coverage denominator. Models reaching no task have no rank, and exact ties share a rank using unrounded values.

At 10%, Gemini 3.7 Flash reaches all 46 tasks using an average of 5.4% of the budget, and Opus 5 reaches the same coverage using 24.5%. At 80%, Opus 5 reaches 35 tasks and Gemini 3.7 Flash reaches 15. Fast initial progress and broad completion at demanding targets therefore favor different models.

Pass rate $\geq 10\%$

Rank	Model	Reached	Budget time (min)	Budget percent (%)
1	**Gemini 3.7 Flash**	**46**	**19.4**	**5.4**
2	Grok 4.6	46	22.0	6.1
3	DSV4.1 Flash	46	26.2	7.3
4	GPT-5.6 Sol	46	30.2	8.4
5	GLM-5.3	46	54.3	15.1
6	Hy4 preview	46	72.0	20.0
7	Opus 5	46	88.3	24.5
8	DSV4 Pro 0813	45	58.0	16.1
9	Kimi K3	45	87.8	24.4
10	DSV4 Flash 0731	44	27.1	7.5
11	Sonnet 5	44	57.1	15.9
12	Qwen 3.8 Max	42	74.1	20.6

Pass rate $\geq 50\%$

Rank	Model	Reached	Budget time (min)	Budget percent (%)
1	**Opus 5**	**43**	**113.0**	**31.4**
2	Grok 4.6	42	46.3	12.9
3	GPT-5.6 Sol	37	57.4	16.0
4	DSV4.1 Flash	35	61.2	17.0
5	Kimi K3	35	129.1	35.8
6	Hy4 preview	34	105.6	29.3
7	GLM-5.3	34	112.1	31.1
8	Gemini 3.7 Flash	32	53.8	15.0
9	Sonnet 5	29	109.0	30.3
10	DSV4 Pro 0813	27	91.5	25.4
11	Qwen 3.8 Max	25	110.9	30.8
12	DSV4 Flash 0731	24	39.4	10.9

Pass rate $\geq 80\%$

Rank	Model	Reached	Budget time (min)	Budget percent (%)
1	**Opus 5**	**35**	**153.4**	**42.6**
2	Grok 4.6	29	92.1	25.6
3	Kimi K3	18	168.6	46.8
4	DSV4.1 Flash	17	93.5	26.0
5	GPT-5.6 Sol	16	62.7	17.4
6	Hy4 preview	16	135.3	37.6
7	Gemini 3.7 Flash	15	81.8	22.7
8	Sonnet 5	15	142.9	39.7
9	GLM-5.3	13	141.5	39.3
10	DSV4 Pro 0813	11	140.4	39.0
11	Qwen 3.8 Max	10	150.3	41.7
12	DSV4 Flash 0731	6	94.9	26.4

Pass rate $\geq 95\%$

Rank	Model	Reached	Budget time (min)	Budget percent (%)
1	**Opus 5**	**20**	**218.3**	**60.6**
2	Grok 4.6	7	90.9	25.2
3	GPT-5.6 Sol	5	131.6	36.5
4	Kimi K3	4	117.2	32.6
5	GLM-5.3	4	149.1	41.4
6	DSV4.1 Flash	4	153.1	42.5
7	Hy4 preview	4	177.7	49.4
8	Gemini 3.7 Flash	3	53.0	14.7
9	DSV4 Flash 0731	1	86.2	23.9
10	Sonnet 5	1	100.6	27.9
11	DSV4 Pro 0813	1	171.8	47.7
–	Qwen 3.8 Max	0	–	–

Table 6: Coverage and first-crossing costs at fixed completion targets. Reached counts are out of 46 tasks. Both costs average over reached tasks. Panels rank by count (descending), then budget time (ascending). Bold marks the leader. A dash denotes an undefined rank or mean.

C ADDITIONAL TASK TRAJECTORIES

Figure 6 compares progress across tasks and systems. Each curve shows the highest hidden-test pass rate reached so far, highlighting improvements and plateaus. The six tasks were selected for slow progress among tasks on which at least one system passed more than three quarters of the tests. The panels share an effective-time axis and a model legend.

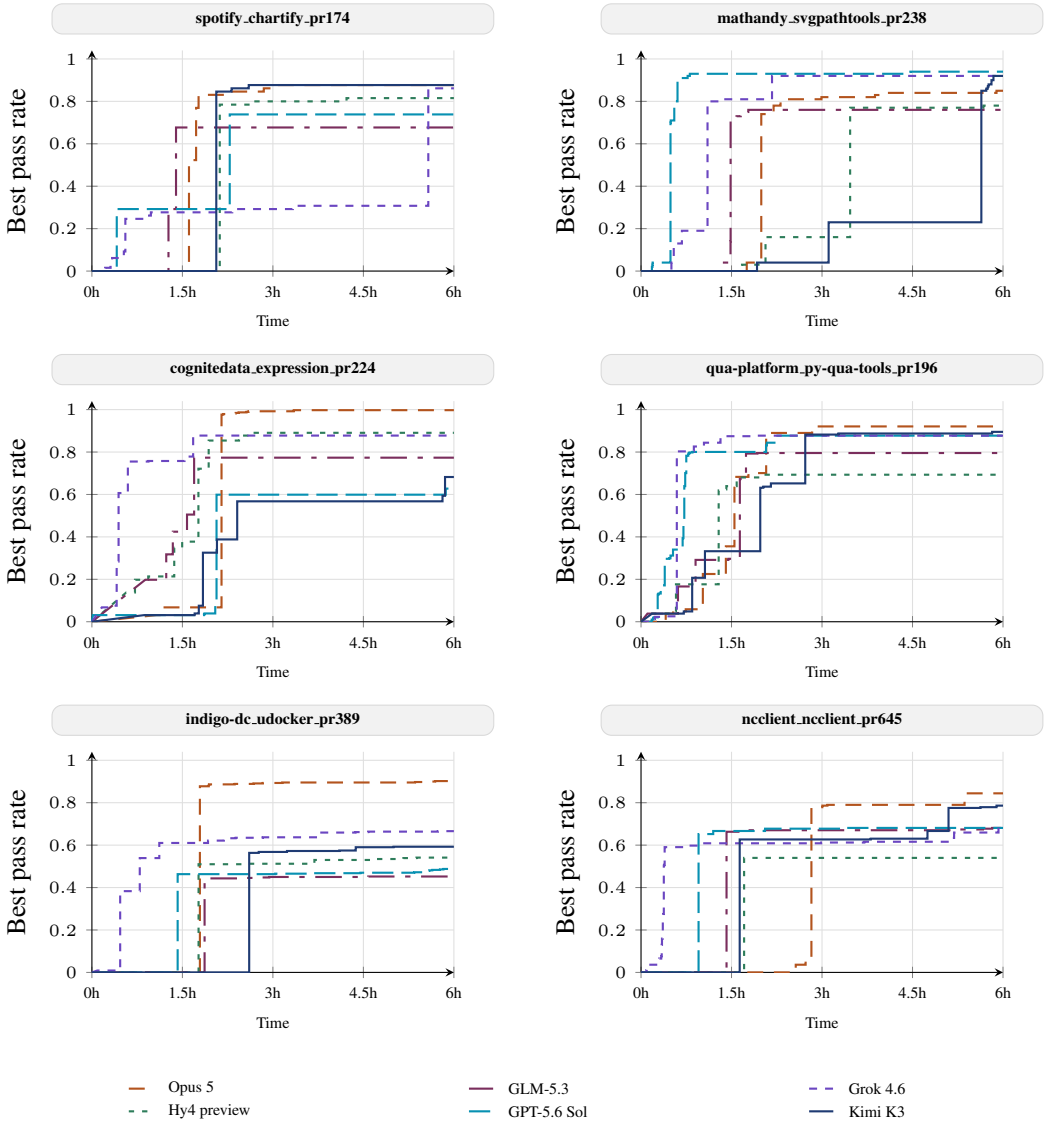


Figure 6: Best-pass-rate trajectories for six tasks. Each curve shows the highest valid pass rate observed so far. The horizontal axis shows budget time, dq/B, where q is effective consumed tokens, B is the token ceiling, and d is the common six-hour reference duration. The panels share a model legend.

C.1 Lift attribution across all models

Labeling procedure. A lift occurs when a scored checkpoint exceeds the trial's previous best pass rate, starting from zero. Lift occurrence, magnitude, and budget position are computed from recorded test results and execution metadata. Benchmark scores and rankings are computed independently of the behavioral annotations.

We use DSV4.1 Flash as an LLM annotator to classify the associated code change as added functionality, a repair to satisfy the specification, or another type of change, and to identify its apparent information source. The evidence includes code diffs, changed test outcomes, the specification, and agent steps. A separate LLM pass assesses feedback attribution, requiring a recorded link between a scalar reading and the subsequent investigation or hypothesis behind the change. Targeted checks examine the supporting evidence for individual annotations. The annotators can see the scores and test results. The analysis covers 5,481 lifts across 12 agent systems.

These annotations provide an exploratory interpretation of changes accompanying observed gains. Agreement with independent human annotations has not been systematically measured, leaving the accuracy of individual labels and aggregate category shares uncertain. Category shares should therefore be interpreted conditional on the annotation procedure. Estimating how often agents use each behavior or its causal effect requires separate analysis.

C.1.1 ANNOTATION PROMPTS

The prompts below summarize the annotation decisions using the categories displayed in the figures.

Lift classification.

```
For each lift included in the plots, inspect the code diff, changed test
outcomes, specification, and agent steps. Assign one change type:

Function added: adds previously missing functionality.
Contract fix: repairs existing behavior.

Return lift, change_type, and supporting evidence.
```

Information-source attribution.

```
Assign one information source to each plotted lift:
Specification, Self-test, or Feedback.

Use Feedback when a recorded scalar reading prompted the investigation
or hypothesis behind the change, including when a self-written test later
confirmed it. Mere proximity to a feedback call is insufficient. Base the
decision on the agent's recorded actions and available readings.

Return lift, source, and supporting evidence.
```

The plots show function additions and contract fixes with available trajectories, dividing each model's gains by the full task count. Figure 7 extends Figure 4's gain and composition panels to all models in Table 1 order, with the same categories, budget intervals, and averaging rule.

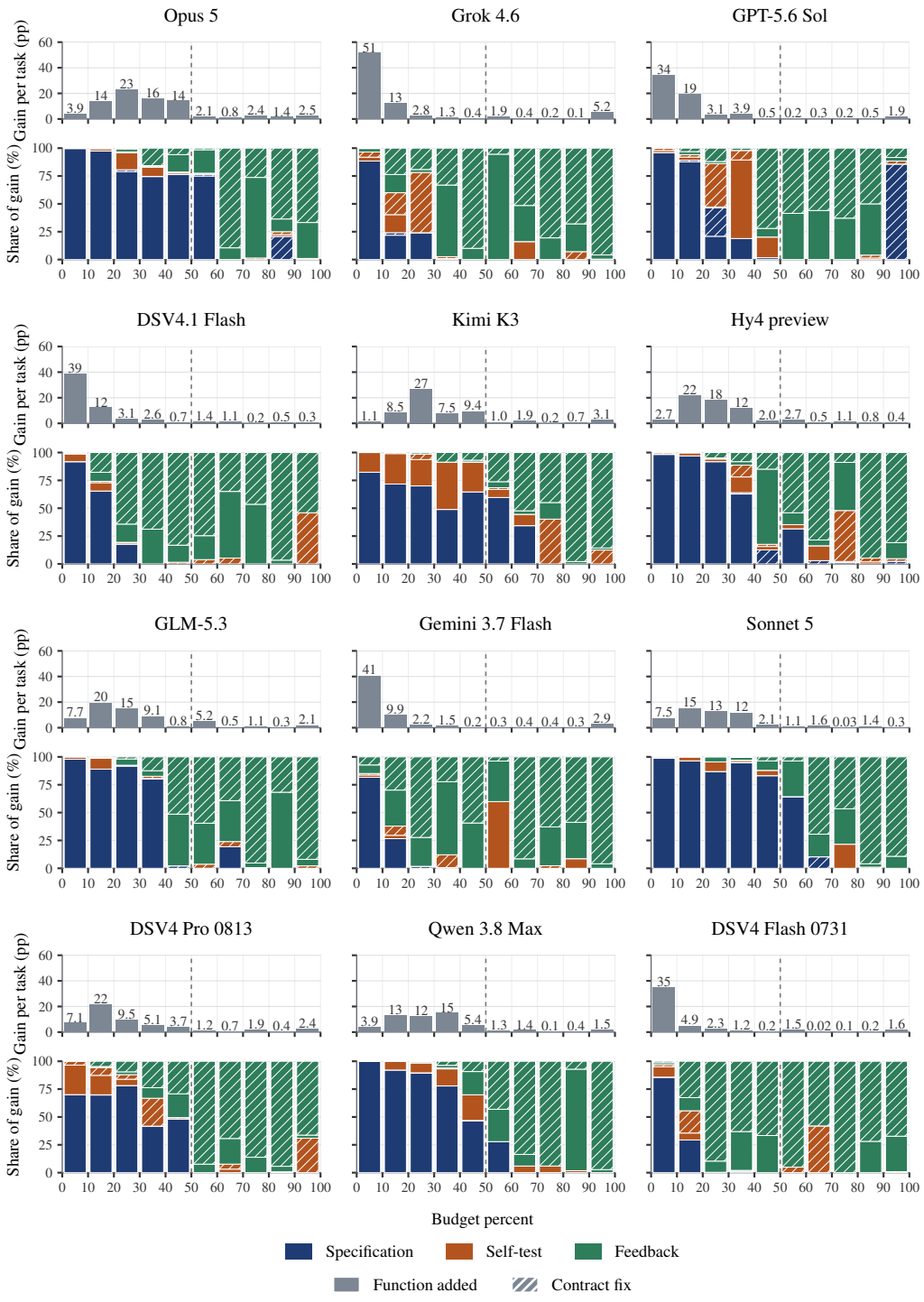


Figure 7: Lift attribution across all 12 evaluated models. Each model has aligned gain and composition panels, as in Figure 4. Upper panels sum pass-rate gains from function additions and contract fixes in 10%-wide budget intervals, divided by all 46 tasks per model, on a common scale. Lower panels show each bin's gain composition. Colors identify the information source and hatching marks contract fixes. Other change types are excluded. Dashed lines mark half the budget. Lift types and information sources use exploratory LLM-assisted annotations, with validation limits described above.

D SENSITIVITY ANALYSES

D.1 METRIC DISCRETIZATION SENSITIVITY

The grid in Section 4.2 groups fine-grained pass-rate differences within completion bins. The underlying pass rates still reflect differences in test density across features. To check sensitivity to this discretization, let $p_i \in [0, 1]$ be task i's best pass rate over N tasks and compare A with

$$Q = \frac{1}{N} \sum_{i=1}^{N} p_i^2 = \int_0^1 2t \, \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[p_i \geq t] \, dt.$$

This is the continuous linearly weighted reach-rate AUC. Division by the shared task count preserves the ranking of the squared sum. Across the twelve systems, the full-budget ranking is unchanged (Spearman $\rho = 1.000$). Table 7 reports both scores. This comparison checks threshold discretization under the same completion-weighting preference.

D.2 SEED SENSITIVITY

Table 8 gives the individual runs behind the interval reported in Section 4.4. Each run's weighted reach-rate AUC is computed from that run's results, and the confidence interval summarizes variation across runs. All ten runs include all 46 tasks, yielding 460 valid completed trials with no missing or duplicate results. Six trials across runs 7–10 completed after resuming from snapshots saved during an infrastructure interruption. Across the 46 tasks the mean spread between a task's best and worst run is 0.308, with 36 tasks spanning more than ten points, 20 more than twenty-five, and ten more than fifty. The per-cell 95% interval averages 0.150 in width over all 46 cells and 0.181 over the 32 cells whose mean best pass rate lies between 0.2 and 0.8, illustrating the task-level variability underlying the aggregate results.

Table 7: Weighted reach-rate AUC A and mean squared best pass rate Q for the same selected trials, ordered by A.

Model	A	Q
Opus 5	0.726	0.767
Grok 4.6	0.628	0.658
GPT-5.6 Sol	0.512	0.539
DSV4.1 Flash	0.485	0.505
Kimi K3	0.472	0.500
Hy4 preview	0.467	0.492
GLM-5.3	0.457	0.475
Gemini 3.7 Flash	0.420	0.442
Sonnet 5	0.392	0.410
DSV4 Pro 0813	0.372	0.390
Qwen 3.8 Max	0.363	0.381
DSV4 Flash 0731	0.311	0.323

Table 8: The DSV4.1 Flash condition re-run ten times over the same 46 tasks, in run order. A is the weighted reach-rate AUC of that run alone and the mean best pass rate is over its 46 trials. The last row reports mean AUC over the ten runs, with the standard deviation of A in parentheses.

Observation	A	Tasks solved	Mean best
run-1	0.4849	1	0.6702
run-2	0.4303	1	0.6268
run-3	0.4963	2	0.6783
run-4	0.5061	1	0.6905
run-5	0.4714	1	0.6648
run-6	0.4952	1	0.6879
run-7	0.4705	1	0.6620
run-8	0.4747	2	0.6704
run-9	0.4657	1	0.6589
run-10	0.4639	1	0.6651
mean	0.4759 (0.0215)		

E FEEDBACK AND ITERATIVE SELF-CORRECTION

The six cases use the same structure: task explanation, chronological agent evidence and actions, our interpretation, and a model or condition comparison with verification. The multi-round cases are followed by a paired Prometheus counter repair from Section 4.6. Quotation marks identify verbatim excerpts from the retained trajectory. Unquoted agent entries are summaries. Opus excerpts are ordinary assistant text, while GLM and Grok summaries may also use retained reasoning fields. Online feedback and local test output are labeled separately. The explanation column contains our analysis. Each case traces feedback through investigation, revision, and verification. Budget shares include other development work.

E.1 GLM REPAIRS OBSERVED MUTATIONS AND PRESERVES CONTAINER BEHAVIOR

Task explanation. The `properties` library lets an object validate changes to a stored list, set, or dictionary and notify listeners. With mutation observation enabled, an operation such as appending to a list must retain the new value through property reassignment. The wrapper must also preserve the caller's concrete container type. For an `OrderedDict`, callers should retain its entries, ordering, and methods such as `move_to_end`. These requirements interact because constructing or copying a wrapper can itself dispatch through overridden mutation methods.

Agent reasoning / output	Action	Explanation (authors)
GLM-5.3. Diagnose lost updates. After a feedback gain from another repair, the agent expands its local checks. An in-place set update silently loses the added element.	Traces the operator's return value through Python's assignment back to the property. Changes list/set/dict operators to return the updated value.	Returning the old wrapper overwrites the reassigned value. The local test isolates the difference between updating and returning the container.
GLM-5.3. Measure and extend. The operator checks pass and online feedback increases. The agent explicitly continues testing mutation observers.	Tests other mutators, then preserves concrete subclasses when creating observable wrappers. Checks lists, sets, and `OrderedDict`.	Correct in-place updates do not establish subclass preservation. The next revision addresses a second part of the same container contract.
GLM-5.3. Correct the new regression. Feedback improves after the subclass change, but the local `OrderedDict` test shows empty contents.	Follows construction and copying through overridden methods. Fills the wrapper using unbound base-container operations, then reruns the controls.	A higher aggregate score coexists with a local regression. The agent uses the failing local check to complete the revision before accepting it.
GLM-5.3. Verify retained behavior. Local checks preserve entries, order, and subclass identity. The next feedback retains the preceding gain.	Commits the copy fix and checks detached wrappers and custom container subclasses during subsequent regression testing.	The final revision has local confirmation with unchanged immediate feedback. The earlier operator and class-retention gains persist.

Model comparison / outcome. GLM's successive operator, subclass, and copy revisions occur at 42.0%, 46.0%, and 46.7% of its calibrated budget. The matching retained tests confirm the first two gains. The last revision repairs the locally observed empty-container regression without an additional score increase. DSV4 Flash 0731's retained wrapper preserves values in the ordinary operations but converts an `OrderedDict` to a plain dictionary subclass. Its caller consequently loses `move_to_end`, which raises `AttributeError` in our same-input check. Repaired GLM retains that method and the expected order. The diagnostic executes the actual wrapper, copy, and mutation code with a controlled property owner. Full listener and nested-validation behavior is outside its scope. The revisions' scoring boundaries have unchanged test-record coverage and no evaluator timeouts.

E.2 Opus and Sonnet complete command parsing through integration tests

Task explanation. Cleo builds command-line applications. A registered command `env set` takes an environment name. For `env set production`, the shell supplies separate words, and the framework must combine the command name while preserving `production` as its argument. Both models initially mishandle this boundary. Internal calls and help requests also need to preserve the complete registered command name.

Agent reasoning / output	Action	Explanation (authors)
Sonnet 5. Build and self-test. Local greeting, option, standard-input, and application-dispatch checks succeed. The agent describes a working functional core.	Implements test helpers, checks ordinary command execution, then requests feedback and continues implementation.	These checks support broad functionality. The command-level helper quotes multiword names, performing normalization that the application lacks.
Sonnet 5. Respond to flat feedback. Online feedback remains at 175/227. *"Let me write one comprehensive smoke test"*	Broadens checks and initiates a focused application/command audit.	The agent responds to repeated unchanged feedback by expanding test scope beyond the earlier targeted checks.

Continued on the next page.

Agent reasoning / output	Action	Explanation (authors)
Sonnet 5. Find and repair the public path. Local application testing fails to execute a multiword command. The agent identifies separate name tokens reaching the argument binder.	Combines the name before binding. Checks multiword commands with and without trailing arguments, including `env set production`.	The application now supplies the same command-name unit that the test helper had supplied automatically. The trailing value remains an argument.
Sonnet 5. Extend a confirmed diagnosis. Online feedback rises to 176/227. The agent asks whether `Command.call()` has a related argument collision.	Finds an omitted command-name token in internal dispatch. Repairs it, verifies ordinary and multiword targets, and reruns regression checks.	The measured gain prompts a concrete follow-up hypothesis. Local output confirms correct argument delivery; the next feedback stays unchanged.
Sonnet 5. Check another entry point. Local output for `help env list` says that `env` does not exist. The agent identifies the missing normalization for the help target.	Combines the help target, verifies the displayed help, commits the repair, and requests feedback again.	The help command's argument is itself a command name. This repair has local validation, with feedback still at 176/227.
Opus 5. Comparator repair. An application test loses the trailing value. *"The namespaced-command argv rewrite drops the trailing argument."*	Restores the script-name slot expected by its input constructor. Checks arguments, options, and the script name.	Opus's initial normalization attempted the right grouping but rebuilt the input incorrectly. Its repair preserves the same public-input property.

Model comparison / outcome. Opus repairs the public entry point at 40.3% of its budget; Sonnet does so at 88.2%. Both retain correct argument handling there. Sonnet's single-file public-entry repair changes the matching retained test from failure to success, and its independent score and online feedback agree on 175/227 to 176/227. Its subsequent internal-call and help repairs have local confirmation with unchanged immediate feedback. Opus's repair commit also includes UI changes. The budget coordinates include other work and do not measure parsing-only effort.

E.3 GPT REPAIRS A TEXT ENCODER AND ITS PAIRED DECODER IN SUCCESSIVE ROUNDS

Task explanation. A monitoring library writes a metric's human-readable HELP description into a text stream and reads it back. Its OpenMetrics format escapes quotes, backslashes, and newlines. A description containing quoted text must be encoded as required and recovered without extra backslashes. The library also supports Prometheus text, whose HELP escaping rules differ. A pair of helpers that agree only with each other can still fail to exchange text with a conforming producer or consumer.

Agent reasoning / output	Action	Explanation (authors)
GPT-5.6 Sol. Repair output. The agent identifies that OpenMetrics HELP output leaves quotes unescaped despite the format's contract.	Uses the OpenMetrics string-escaping helper, tests a description with quotes, a backslash, and a newline, and commits the isolated change.	This fixes the outgoing representation. The test checks the actual emitted text against the required escaping rule.
GPT-5.6 Sol. Use feedback to choose the next repair. Online feedback rises to 276/374. The agent notes that the paired parser still preserves escaped quotes literally.	Adds an OpenMetrics-specific decoder and applies it to HELP metadata, keeping the other text format's rules separate.	The confirmed output repair leads directly to testing the reverse path. A shared helper had applied one format's rules to both formats.
GPT-5.6 Sol. Verify the pair. An encoder-to-parser check now recovers the original description. The subsequent clean feedback reading is 277/374.	Commits the decoder and checks the exact description after a full encode/parse round trip. Continues to related metadata handling.	Independent exposition and parser tests each change from failure to success at their respective revisions.

Continued on the next page.

Agent reasoning / output	Action	Explanation (authors)
Qwen 3.8 Max. Comparator regression. The agent reasons that HELP text should leave quotes unescaped. Its broad smoke test uses a simple description without these characters.	Applies a quote-free HELP helper to both output formats and retains a decoder that also leaves escaped quotes unchanged.	The two helpers remain internally consistent on their own output while violating the OpenMetrics representation and decoding contract.

Model comparison / outcome. GPT's encoder and decoder revisions land at 53.9% and 54.8% of its budget. The retained independent scores advance from 275/374 to 276/374 and then 277/374, with unchanged parsed-test coverage and zero timeouts at both boundaries. The final online reading is taken on the unchanged decoder commit after earlier attempts lose their terminal output. Our controlled checks compare emitted text and decoding of the required wire representation separately. Repaired GPT passes both. Qwen's retained helpers leave quotes unescaped and preserve extra backslashes when given correctly escaped input, even though their own round trip passes. The same-input check establishes the codec behavior; it does not attribute the models' whole-task score gap to this property.

E.4 Opus uses feedback to separate local edits from external file changes

Task explanation. Defcon supports font editors with associated image and data files. After loading a file, a user may edit its contents in memory without saving while another program changes the disk file. The detector must report the disk edit and preserve the local value until an explicit reload. A local edit alone should produce no external-change report. This requires storing the last observed disk contents separately from the editable value.

Opus initially compares content after a timestamp change. A later revision removes that gate and compares disk bytes directly with live in-memory data, causing local-only edits to be misreported. Both versions lack a stable disk baseline. The following rounds trace the eventual repair.

Agent reasoning / output	Action	Explanation (authors)
Opus 5. Redirect the search. Local checks of other font operations pass, but feedback stays at 360/393. *"Blind API additions have stopped paying."*	Temporarily disables selected behaviors in its implementation, measures feedback, and restores the code between comparisons.	The agent changes its search strategy after repeated flat feedback. Candidate perturbations test where working behavior affects the score.
Opus 5. Interpret the comparisons. Disabling detection plus reload gives the same score as disabling reload alone. *"Let me revert and hunt for what makes it fail."*	Investigates external-change detection. Fixes an unsaved-layer case, then continues testing image and data files when feedback remains unchanged.	Opus treats the scalar comparison as a clue that detection may already be failing. The comparison alone does not establish test coverage.
Opus 5. Reproduce and revise. A local-only edit produces an external-change report. *"Let me stamp the on-disk snapshot separately from live data."*	Adds disk snapshots, preserves them during assignment, and refreshes them on load/save. Checks local edits, disk edits, and reload.	The baseline must remain stable while the user edits. Current disk content is compared with the snapshot, preserving the distinction between states.
Opus 5. Verify and extend. Online feedback rises to 381/393. The agent recognizes the gain and checks for further false positives.	Retains the repair and tests deletion and local changes across the font.	Independent scoring confirms the same commit-level increase. Subsequent checks apply the state distinction to related operations.
GLM-5.3. Comparator judgment. Earlier reasoning considers a file changed locally and on disk: *"Edge case, skip is fine."* Later feedback from disabling detection shows no decrease.	Keeps the dirty-file skip rule. After considering absent coverage and already-failing code, turns to notification handling.	The retained guard prevents checking genuine disk changes whenever local edits coexist. A separate disk value is stored but bypassed.

Model comparison / outcome. On the same combined-edit input, repaired Opus reports the external modification and GLM misses it. Both handle local-only and disk-only edits correctly. Our controlled

load, assignment, scan, and reload checks also cover unchanged files and the same new value written both locally and on disk. Opus handles all these states; GLM misses both combined-edit cases. These are author-run diagnostics with a controlled reader, separate from the agent's recorded tests.

The Opus repair lands at 97.9% of its calibrated budget and remains in the final code. Online feedback and independent scoring both move from 360/393 to 381/393. The commit changes the image and data containers. Both scoring boundaries retain complete test records with no evaluator timeouts. The gained tests include container and font/layer integration cases, so the score difference is attributed at commit level. The controlled diagnostic establishes the specific data-state property.

E.5 GROK REPAIRS METADATA COMPARISON AND THEN ITS BYPASSING CONTROL FLOW

Task explanation. DVC's data index compares file entries to decide whether they changed. An entry contains a content hash and metadata, such as size, identity, and executable status. With default comparison, matching hashes alone are insufficient: a permission change can matter even when the file bytes are identical. The API also offers explicit metadata-only and hash-only modes. A caller-supplied metadata projection can intentionally restrict which metadata matter.

Agent reasoning / output	Action	Explanation (authors)
Grok 4.6. Revisit a stalled comparison. After repeated flat feedback, the agent identifies that its metadata helper compares only a selected field tuple.	Changes the default helper to compare complete metadata objects, retaining the caller's explicit projection override. Commits and requests feedback.	Fields omitted by the tuple can cause a changed file to appear unchanged in metadata-only mode. The full-object comparison restores the contract.
Grok 4.6. Verify the first revision. Online feedback reaches 120/168, and the agent recognizes the gain.	Keeps the helper change and continues reviewing index behavior.	The matching metadata-only test changes from failure to success. The default path still contains a separate shortcut that bypasses metadata.
Grok 4.6. Repair the remaining path. Later, the agent identifies that entries with hashes return early from classification without checking metadata.	Removes the shortcut so default comparison requires matching metadata and hash results. Commits the revision and later requests feedback.	Repairing the comparison helper had not repaired every caller. This revision makes the corrected rule reachable in default comparison.
Grok 4.6. Confirm progress. Online feedback is 123/168. The agent acknowledges another gain and continues checking remaining contracts.	Retains both revisions.	Independent scoring isolates a matching combined-comparison test gain at this commit. The online interval also contains other index work.

Model comparison / outcome. Grok's metadata-helper and default-path repairs occur at 40.5% and 64.8% of its budget. Sonnet 5 handles the tested comparison modes from its first implementation at 34.0% and retains them. Controlled checks confirm that both final classifiers recognize metadata changes under matching hashes and respect explicit hash-only and metadata-projection controls. Grok's initial classifier already handled the combined case; a later hash-first shortcut introduced the bypass that the second repair removes. The case therefore distinguishes a repaired omission from a repaired regression. Both scored repair boundaries have zero timeouts and unchanged parsed-test coverage. The comparison concerns the stated file-entry rules; the cumulative budget includes other work and the helper diagnostics do not establish whole-index traversal correctness.

E.6 DSV4.1 FLASH CORRECTS COUNTER NAMES LEFT UNCHANGED AT VOLUNTARY COMPLETION

Task explanation. Prometheus's text parser converts metric samples into Python objects. For a declared counter named `a`, the supplied specification explicitly requires family name `a` and sample name `a_total`. Parsing the following input with `text_string_to_metric_families` must normalize the sample name:

```
# TYPE a counter
a 1
```

Both conditions use DSV4.1 Flash with the same harness and reasoning configuration. The single-pass final parser accepts the counter sample and appends it with its original name `a`. The main run initially does the same. Its repair adds the following branch before appending the sample:

```
if current_type == 'counter':
    internal, _ = _family_details(current_raw_name, current_type)
    if sample_name == internal:
        sample_name = internal + '_total'
```

Agent reasoning / output	Action	Explanation (authors)
DSV4.1 Flash. Inspect the parser. After a grouping experiment leaves feedback unchanged, the agent asks: *"Our family a sample name `a`; maybe expected sample name `a_total`?"*	Selects counter-name normalization as the next hypothesis and changes the branch that appends a sample to its declared family.	The candidate implements a rule already stated in the specification. The scalar supplies an aggregate check on the proposed change.
DSV4.1 Flash. Evaluate and retain. Online feedback increases from 45.99% to 49.20%. The agent says *"Keep this experiment"* and then cites the specification's counter-naming rule.	Retains the normalization and reasons through the treatment of inputs whose sample names already end in `_total`.	The observed feedback informs acceptance of the revision. The corrected branch remains in the final implementation.

Condition comparison / outcome. Our same-input diagnostic executes the retained parser code with shared in-memory metric containers and identifier validators. The reference and pre-repair main code return sample name `a`; the repaired and final main code return `a_total`. A gauge control retains `a` in every implementation. The upstream implementation at the task's fixed parent commit also returns `a_total`. Independently scored checkpoints show `test_simple_counter` changing from failed to passed at the repair and remaining passed at the end, with unchanged test coverage and no evaluator timeout at the repair boundary. The comparison establishes this specified parser behavior using a direct code probe, independently of omitted export interfaces exercised elsewhere in the task. Its cross-condition interpretation follows the budget, continuation, and scoring boundaries in Section 4.6.

Evidence boundaries. The cases were selected after examining outcomes. The agent-evidence column contains attributed excerpts or summaries, the action column records observed work, and the explanation column supplies our interpretation. Online feedback, local tests, independent scoring, and author-run diagnostics have separate roles. Isolated edits support narrower attribution; mixed commits retain the limits described above. The multi-round cases contain successive substantive revisions separated by feedback; the Prometheus pair traces a single retained repair against a no-feedback reference. Their local and scored outcomes are stated separately. The cases explain feedback-guided engineering work without estimating behavior prevalence or its aggregate causal effect.