SWE-CLIMB: BENCHMARKING CODING AGENTS ON REAL WORK IN MILLION-LINE BROWNFIELD CODEBASES

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are moving from writing new code to maintaining the large, long-lived systems that software already runs on. In such brownfield codebases, the hard part of a small change is knowing the system: where the entry point lives, which configuration rule wins, and which undocumented behavior users depend on. Existing benchmarks mostly judge whether a final patch passes tests, collapsing this knowledge into a single pass/fail signal that cannot show what an agent knows or misses. To measure it directly, we introduce SWE-CLIMB, which measures maintenance as a climb of three rungs: *explaining* how a change flows through the system, *operating* its existing toolchain, and *evolving* it without breaking old behavior, on real repositories whose median size exceeds a million lines of code. Evaluating six frontier agents, we find that knowing is not operating: agents with similar explanation scores can differ sharply in running a system. Agents also know how to change a system better than what must not change: all six recover legacy invariants less reliably than fix paths, and a single broken old behavior can void an otherwise near-complete patch. The required knowledge is system-specific, as removing repository access cuts exploration scores by 62–76%. Finally, evaluating brownfield work is itself a brownfield problem: real repositories make graders fragile, and we contribute a requirement-grounded grading contract under which every zero must have a cause.

1 INTRODUCTION

Most software engineering modifies existing systems rather than writing greenfield code (Lehman, 1980; Winters et al., 2020), and coding agents increasingly shoulder this work. On public GitHub, merged pull requests carrying default Claude Code attribution surged from near zero in early 2025 to 13.4% by August 2026 (Figure 1b), a lower bound given uncredited runs and competing tools (GitHub, 2025; OpenAI, 2025; Google, 2025b). Meanwhile, the systems receiving these changes continue to expand: Kubernetes has grown tenfold since 2015, and MongoDB over thirtyfold since 2010 (Figure 1a).

In such systems, the hard part of a small change is knowing the system. A feature may depend on an entry point several modules away, a configuration precedence rule, and a compatibility branch that no document promises but users depend upon. Under Hyrum's Law (Wright, n.d.), every observable behavior eventually acquires dependents, so the maintainer must know not only how to make the change but also what must not change. We call this *task-relevant system knowledge*: the entry points, integration mechanisms, precedence rules, invariants, and verification paths needed for one concrete goal. It is scattered well beyond the edit; in our Exploration tasks, the median reading set spans 18 source files across two to three top-level directories. We use *brownfield* for this accumulated history, scale, and implicit contract surface, without judging code quality. As agents take on maintenance, the bottleneck shifts from generating code to knowing the system.

In such brownfield systems, the core difficulty of a small change is knowing the system. A single feature often hinges on distant entry points, obscure configuration precedence, and implicit compatibility branches. Under Hyrum's Law (Wright, n.d.), every observable behavior acquires dependents, so maintainers must know not only how to make a change, but also what must not break. We call this

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.

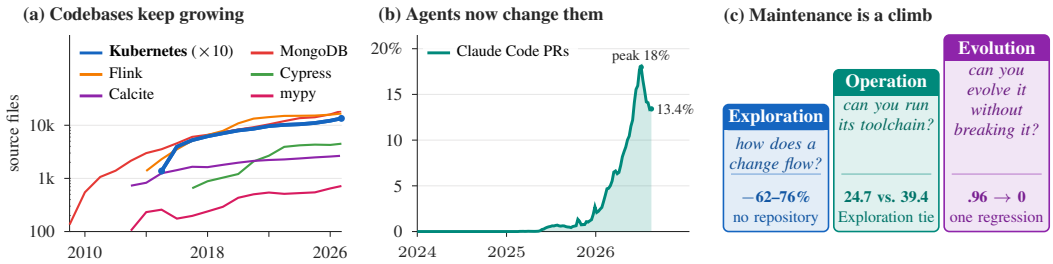


Figure 1: **Agents are being handed systems that keep growing.** (a) Tracked source files in six SWE-CLIMB repositories (log scale). (b) Share of merged public GitHub PRs with the default Claude Code attribution, a lower bound on agent use (Appendix G). (c) One headline result per rung: Exploration scores drop 62–76% without the repository; two agents nearly tied on Exploration score 24.7 vs. 39.4 on Operation; a single pass-to-pass regression zeroes a .96 Evolution patch.

task-relevant system knowledge. This context scatters far beyond the edit itself: in our Exploration tasks, the median reading set spans 18 source files across two to three directories. As agents take on maintenance, the bottleneck shifts from generating code to acquiring this system knowledge.

SWE-CLIMB frames maintenance as a three-rung climb (Figure 1c; Figure 2). **Exploration** evaluates whether an agent understands how a change flows through the system, graded against 2,404 atomic gold facts. **Operation** tests whether it can drive the real toolchain to produce checkable results. **Evolution** requires implementing an upstream change while pass-to-pass tests veto any regression on legacy behavior. Each rung raises the commitment: from saying, to doing, to doing without breaking. Across 130 repositories, SWE-CLIMB spans 120 Exploration, 120 Operation, and 57 Evolution tasks.

Evaluating six frontier agents yields four findings. **Knowing vs. operating.** Doubao and GPT-5.5 virtually tie on Exploration at 68.1 and 69.4, yet diverge by 14.7 points on Operation at 24.7 and 39.4, proving that single-rung evaluations misrank maintenance ability. **Preservation blind spot.** Legacy invariants are universally recalled worse than fix paths, and a single broken behavior routinely vetoes an otherwise 96%-passing patch. **System specificity.** Removing the repository slashes exploration scores by 62% to 76%, while 42% of rubric-item instances receive less than full credit even when agents inspect all reference files. **Grading contracts.** SWE-CLIMB admits only prompt-grounded assertions, accepts diverse valid implementations, and preserves agent rankings across judge swaps with a Spearman correlation of 0.83.

Our contributions are:

- **A benchmark separating the three rungs of maintenance.** SWE-CLIMB isolates explaining, operating, and evolving across 130 legacy codebases with a median of 1.63M lines, evaluating each capability on its own rather than collapsing them into a single pass/fail signal.
- **A fact-level representation of task-relevant knowledge.** We formalize maintenance context as 2,404 weighted atomic facts backed by equivalence classes of source evidence, rewarding genuine system understanding instead of recall of a single rigid file list.
- **A requirement-grounded grading contract.** We enforce that all hidden assertions follow from the prompt, structurally diverse valid answers pass, and every Evolution failure is attributed to the patch, yielding stable rankings across judge swaps with a Spearman correlation of 0.83.

2 SWE-CLIMB: TASKS AND SCOPE

2.1 TASK-RELEVANT KNOWLEDGE AND REPOSITORY BACKGROUND

A gold fact must be necessary for the stated requirement and recoverable from the agent-visible snapshot and prompt. Facts include true entry points, registries and dispatchers, configuration precedence, old-behavior invariants, and build or test paths. Repository trivia and private helper names known only from the eventual upstream patch are excluded. Evidence is represented as equivalence

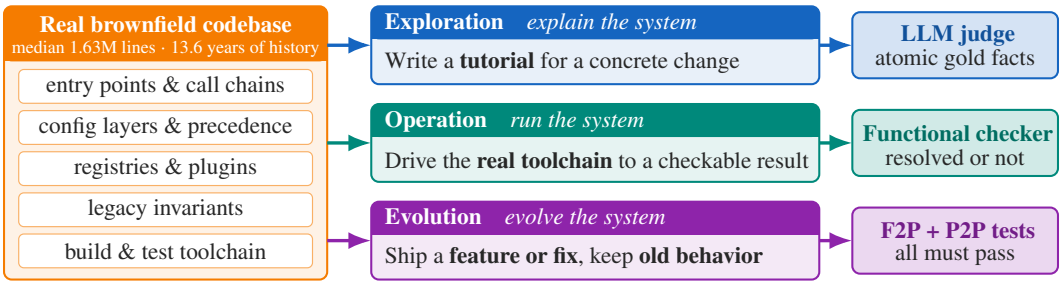


Figure 2: **Task design.** Each track pins one deliverable to a frozen, de-identified, history-free snapshot. The facts an agent needs (left) lie far beyond the eventual edit; each track is graded on its own scale.

classes: source, tests, or configuration may establish the same fact, so there is no single mandatory file list. Difficulty comes from natural repository evolution; we do not inject synthetic smells or combine unrelated changes to manufacture hardness.

The 130 canonical upstream repositories span 20 primary languages. Their median profile is 1.63M physical code lines, 13.6 years of history, 49.1k commits, and 1,406 contributors. The 54 repositories behind the 57 Evolution tasks are of the same scale, with a median of 1.34M code lines, 11.3 years of history, 22.3k commits, and 554 contributors (Appendix F). Exploration reading sets span a median 18 source files and two to three top-level directories, with at least ten files in 111 of 120 tasks. Ecosystem-level adoption and declared-debt proxies are in Appendix G.

2.2 Three Task Interfaces

To diagnose where agents fail during brownfield maintenance, SWE-Climb translates the conceptual climb into three concrete task interfaces. Monolithic patch benchmarks conflate distinct modes of failure: an agent that produces an invalid patch may have missed the architectural entry point, failed to execute the local toolchain, or introduced an accidental regression into existing behavior. By decoupling these stages, each interface pins the agent to a frozen, de-identified repository snapshot while enforcing a distinct deliverable and evaluation contract. The three interfaces form a ladder of increasing commitment, scaling from architectural explanation to toolchain operation, and finally to safe end-to-end code evolution (Figure 3).

Exploration: explain a concrete change. Given a repository snapshot and a behavioral goal, the agent writes a structured developer guide for an incoming engineer. The guide must detail the current system execution path, integration points, legacy invariants, and verification strategies, while source patches are strictly prohibited. To prevent evaluation from collapsing into rigid lexical matching, each task is graded against atomic gold facts partitioned into importance tiers of 5, 3, or 1 points. Across 120 Exploration tasks, we curate 2,404 atomic facts (median 18 per task), each substantiated by equivalence classes of source evidence across code, tests, and configuration.

Operation: run the existing system. The agent receives a runtime objective, a snapshot, and required test fixtures, and must author an executable configuration file, driver script, or command sequence that steers the repository's real toolchain to a verifiable state. The prompt defines the input and output contracts but strictly withholds the configuration mechanism. Because modern build and runtime systems enforce intricate precedence rules, naming the relevant configuration layer is insufficient; the submitted artifact must successfully execute within the repository environment.

Evolution: evolve the system safely. An Evolution task freezes a pre-change base commit for an upstream feature, bug fix, or behavior-preserving refactoring. The agent must implement the required behavior directly within the codebase without breaking historical guarantees. Grading couples new functionality with regression defense: fail-to-pass (F2P) test cases verify the intended change, while extensive pass-to-pass (P2P) test suites immediately veto any solution that breaks pre-existing behavior.

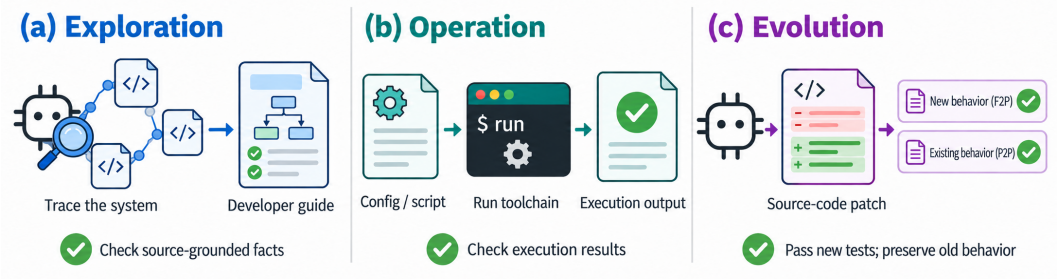


Figure 3: **Three task interfaces.** Exploration explains the system; Operation runs its toolchain; Evolution changes its functionality while preserving existing behavior.

2.3 Worked task examples

An Exploration task in MongoDB asks how retryable-write state remains consistent after a replica returns from a diverged history. The prompt strips revealing keywords like rollback and oplog, requiring the agent to identify a recovery hook and field-level predicates within one 70KB C++ file among 2,783 candidates. The reference represents verifiable architectural facts rather than an instruction to mimic an upstream patch. In Cypress, an Operation task requires an executable script that resolves effective project configuration under strict precedence rules: two files among 6,537 dictate that command-line options override configuration blocks.

On the Evolution rung, a Kubernetes task requires implementing in-place resource resizing for running Pods under an alpha feature gate while maintaining conservative node-level resource accounting. The prompt specifies twelve behavioral requirements without disclosing any file names, types, or function signatures, whereas the reference patch spans 73 files across 12,283 Go sources. Submissions are evaluated against fifty requirement-specific cases alongside fifty pre-existing regression suites, passing only if every case succeeds. Frontier agents successfully recover the wide edit footprint and pass up to 49 of 50 assertions across dozens of files, yet single unhandled clauses or subtle regressions consistently veto their patches, demonstrating that broad coverage alone cannot substitute for invariant preservation.

3 Construction and Evaluation Protocol

3.1 Construction and versioning

Fundamental-track repositories require at least 50k core source lines, 200 source files, 16 months of public history, and 200 closed issues plus merged PRs, together with structural complexity and goals not answerable from the README. Evolution uses million-line-scale repositories. Each task starts from traceable upstream evidence. Builders freeze the base, record its URL, commit, date, and license, and remove Git history, upstream issue/PR links, hidden evaluation assets, credentials, and future implementation clues from the agent workspace. No repository identity contributes more than three fundamental tasks.

A canonical manifest binds the prompt, snapshot, grader, image, model, harness, budgets, and trial index. Any content change creates a new task version, and scores from different versions are never spliced together. Appendices A and C specify the manifest, result states, and construction gates. Solves run without network access; Evolution solvers run on a fenced network whose only egress is the model endpoint, verified before each run.

Requirement-grounded admission. Audits of existing benchmarks have found hidden tests that reject correct answers (OpenAI, 2026b;a) and weak tests or leaked solutions that let incorrect patches pass (Aleithan et al., 2024), and grading against a single reference solution penalizes equally valid alternatives (Gorinova et al., 2026). SWE-Climb builds the corresponding guarantees into admission. Every checker assertion must be traceable to the prompt: key names, formats, orderings, and public surfaces that a hidden check asserts are disclosed, and private choices of the reference are never asserted. An Operation task is admitted only if the reference passes, wrong and empty

answers fail, and a structurally different equivalent answer passes, all on the platform's own scoring images. A no-repository screen revises any task solvable without the source until it requires a repository-specific entry point, precedence rule, legacy branch, or source-defined value. Exploration facts pass a recoverability review by reviewers who never see the upstream change.

3.2 Track-specific evaluation

Exploration. A fixed, tool-free GPT-5.5 judge (2026-04-24 snapshot) sees the rubric, facts, and tutorial, not the repository or trajectory. It returns item-level scores and quoted evidence. The grader aggregates deterministically, applies the documented clamping and penalty rules, and normalizes by the task maximum; it never trusts the judge's reported total. Exploration has no pass threshold.

Operation. Checkers must execute the submission using the repository's own toolchain and assert behavior, with hidden fixtures and anti-hardcoding checks. Passing all assertions yields resolved; a functional failure counts only after the answer has run. Missing answers, non-execution, and infrastructure errors are recorded separately. Every checker assertion must follow from the stated contract (Section 3.1).

Evolution. Each task carries hidden unit tests of two kinds: fail-to-pass (F2P) tests fail on the base and pass on the gold patch, checking the new behavior, and pass-to-pass (P2P) tests pass on both, protecting existing behavior. A task is resolved only if every hidden test passes; a single failing test of either kind leaves it unresolved. Hidden tests must assert public behavior or interfaces explicitly stated in the prompt, rather than incidental private choices in the gold patch. Admission requires base/empty/gold closure (scores 0/0/1), repeatable offline scoring, and P2P sensitivity to mutations of old behavior.

3.3 Scoring contracts and evidence boundaries

The tracks require distinct score semantics. For Exploration, the judge assigns item-level credit on the stated 0/1/3/5 grid and quotes supporting text. Importance weights specify the consequence of omitting a fact: 5 for making the guide unusable, 3 for leaving the implementation fragile, and 1 for completeness. The versioned grader handles undefined items, clamping, and capped hallucination penalties. A fluent answer does not substitute for item-level evidence, and a judge-reported total is retained only for auditing. A normalized score of 70 therefore means partial rubric credit, not that 70% of tasks are solved.

Executable tracks report the binary resolved rate, and Evolution is reported separately from the fundamental tracks.

3.4 Evolution task contracts

Requirements are public; implementation choices are not. An Evolution prompt states user-visible behavior, preservation requirements, and out-of-scope work. Internal paths, private symbols, architecture prescriptions, and editing plans are withheld, so discovering the technical shape of a change is part of the measured work: removing an internal file hint from the mypy prompt lowers its score from .55 to .41. Public surfaces asserted by a hidden test are disclosed: flags, configuration keys, environment variables, wire or SQL fields, user-facing errors and exit codes, and library APIs. A hidden case is admissible only if it follows from that contract and accepts reasonable alternative implementations, and every failing case is attributed against the agent's patch before it counts.

4 Experiments

4.1 Fundamental-track setup and results

Protocol and denominators. Six agents use the same Claude-Code CLI scaffold, system prompt, tool set, and budgets; provider defaults determine sampling. GPT-5.5 uses high effort through a chat-completions endpoint. Each agent runs every task three times. Exploration reports the mean

normalized rubric score over all runs, and Operation reports the resolved rate. Run-level stability is in Appendix I.

Table 1: **Fundamental-track results**, averaged over three runs per task on 120 tasks per track. Exploration is a continuous rubric mean; Operation is the binary resolved rate; both are reported on a 0–100 scale. All agents run in the same Claude Code CLI scaffold with an identical system prompt, tool set, and budget; sampling uses provider defaults, and GPT-5.5 is served through the chat-completions endpoint.

LLM	Reasoning effort	Exploration score	Operation score
GPT-5.5	high	69.4	39.4
Claude Opus 4.8	high	62.8	40.4
Doubao Seed 2.1 pro	high	68.1	24.7
MiniMax M3	high	59.4	25.3
Qwen3.7-max	high	57.5	27.2
DeepSeek-V4 Pro Preview	high	55.9	23.9

Exploring and operating rank agents differently. Doubao is within 1.3 points of GPT-5.5 at exploration (68.1 vs. 69.4), yet GPT-5.5 resolves 39.4% of Operation instances and Doubao 24.7%, fifth of six (gap 14.7 points; paired-bootstrap 95% CI over tasks, 8.3–21.1). Opus, third at exploration, has the highest Operation rate (40.4%). An agent that can describe a change path cannot necessarily drive the toolchain that realizes it, and a leaderboard built on either rung alone would order these agents differently. The two tracks use different scales on different tasks, so we compare orderings rather than magnitudes.

4.2 Evolution-rung setup and results

The Evolution rung contains 57 Evolution tasks, each a real upstream feature or fix in a million-line-scale repository. GPT-5.5, MiniMax, Doubao, and DeepSeek run every task three times; GPT-5.5 runs in the same Claude-Code CLI scaffold as in Section 4. A run's score is the number of resolved tasks divided by 57 and Table 2 reports the mean over the three runs. Solvers run in a container whose only egress forwards the model endpoint, verified before each run and re-checked by trajectory audit.

Table 2: **Evolution-rung results** on the 57 Evolution tasks: resolved rate (0–100), averaged over three runs per task. A task is resolved only if every F2P and P2P unit test passes. GPT-5.5 runs in the same Claude Code CLI scaffold as in Table 1.

LLM	Reasoning effort	Evolution score
GPT-5.5	high	26.9
MiniMax M3	high	8.8
Doubao Seed 2.1 pro	high	8.2
DeepSeek-V4 Pro Preview	high	7.6

Evolution is the steepest rung. The strongest agent resolves barely a quarter of the tasks, and the other three resolve fewer than one in ten. The rung also separates agents more sharply than either fundamental track: GPT-5.5 resolves about three times as many Evolution tasks as any other agent, against a factor of 1.6 on Operation. The ordering shifts again as well. Doubao, a close second to GPT-5.5 at exploration, is third of these four at evolving the system, behind MiniMax.

5 Diagnostic Analysis

5.1 What must not change

Agents know how to change more than what must not change. All six agents have lower recall on legacy-invariant facts than on fix-path facts (Figure 4), with gaps from 2.5 points for MiniMax

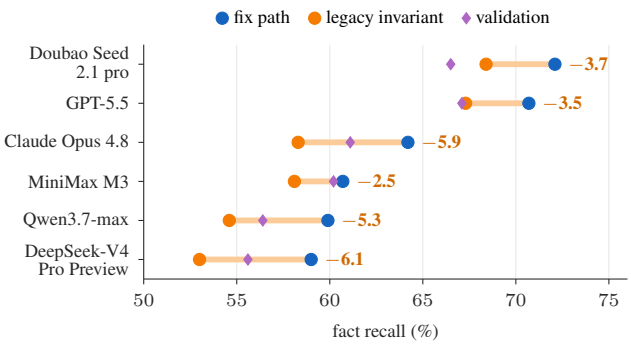


Figure 4: **Agents recall what must not change less reliably than where to change.** Exploration fact recall by fact type over the first two runs, computed from per-item judge scores (sum of awarded points over sum of maxima). Orange numbers give the legacy-invariant minus fix-path gap; it is negative for every agent.

to 6.1 for DeepSeek. Recall is awarded points divided by maximum points for that fact type, before hallucination penalties. The direction holds for every agent, from the strongest explainer to the weakest, which makes preservation knowledge a shared blind spot rather than a property of one model. It reappears wherever old behavior is checked: Operation tasks labeled behavior preservation are among the least resolved (21.9%), and on the Evolution rung a single broken old behavior voids an otherwise near-complete patch: a patch for NetBeans that passes 96% of its new-behavior tests breaks three pre-existing behaviors and leaves the task unresolved. Likewise, a GPT-5.5 patch for Flowable passes all 15 of its new-behavior tests but breaks five pre-existing tests on message start-event subscriptions after a deployment is deleted.

5.2 The knowledge is system-specific

Repository access carries the signal. The no-repository probe keeps prompts, graders, images, and budgets fixed apart from a missing-repository preface and isolation hooks; scorers still use pristine repository copies. GPT-5.5, Doubao, and DeepSeek lose 62–76% of their Exploration score without the repository and 74–87% of their Operation rate (Table 3). The tasks therefore measure knowledge read from the system rather than recalled from general programming experience. Results use clean runs only: a scan of all probe trajectories removes every run that reached a repository copy (Appendix E).

Table 3: **Repository-access ablation on clean runs**, with repository → without repository. With-repository values are the matched comparison subset of the same agents and can differ slightly from Table 1. The probe protocol is in Appendix E.

LLM	Exploration score	Operation score
GPT-5.5	69.8 → 26.4	40.8 → 10.1
Doubao Seed 2.1 pro	69.9 → 22.3	25.2 → 6.6
DeepSeek-V4 Pro Preview	56.5 → 13.5	25.2 → 3.4

Without the source, where to edit is what agents lose. Removing the repository reverses the ordering in the GPT-5.5, Doubao, and DeepSeek probes: fix-path recall drops to 33/30/22, below 39/36/27 for invariants and 50/45/35 for validation. General project knowledge supports preservation and testing advice; locating a particular change path requires reading the system. Access sensitivity also separates agents that absolute scores do not: Doubao gains the most from the repository (about 48 points), so its near-parity with GPT-5.5 at exploring is earned by reading the code.

5.3 FINDING THE CODE IS NOT ENOUGH

Reading the right files matters more than reading many. Across 707 within-model, within-task pairs of Exploration runs, changes in how many gold-fact files an agent reads correlate with score changes nearly three times as strongly as changes in total files read (Appendix I.1). Reading them is still not enough: of 12,374 rubric items for which an agent read every reference file, 42% do not receive full credit (Appendix I.2, Table 9). Validation is often skipped: 29–47% of runs never execute a test, and 98% of GPT-5.5's unresolved runs end by declaring the change complete.

After localization, agents must recover the conditions under which local code applies and the other paths that jointly determine behavior. In Exploration, a GPT-5.5 Neovim run read a dependency between the target window and downstream current-window state but left it out of its explanation (Appendix I.3). In Evolution, GPT-5.5 conflated two Superset states that require different behavior, an unspecified tab restriction and targeted tabs all removed by authorization, and generalized a Presto type conversion from the scalar read path to the separately dispatched batch path (Appendix J.2).

Validation must then exercise the target behavior under the task's conditions. In Operation, Opus 4.8 ran the real Cypress configuration-merge function but not the entry path that first restructures CLI arguments, so the file's `baseUrl` still won and its closing checks never tested that precedence (Appendix I.3). In Evolution, GPT-5.5 identified the TLS and file-transfer risks in Vert.x but executed no behavioral check for them (Appendix J.3). Failures therefore extend beyond localization, to tracing the conditions and paths that determine behavior and checking the resulting output.

5.4 GRADING THAT MEASURES THE AGENT

SWE-CLIMB uses requirement-grounded admission (Section 3.1). We assess Exploration grading stability through repeated scoring and a judge swap.

Exploration scoring shows repeatability and cross-judge agreement. We re-scored a fixed set of 180 submitted tutorials (30 randomly drawn tasks, one submission from each agent) under two conditions. Re-scoring with the same judge changes the pooled mean by under 1 point and individual submissions by a median of 2 points (per-submission correlation .98). Table 4 reports per-agent means when GPT-5.4 replaces the judge.

Table 4: **Exploration judge swap** on 180 fixed submissions (30 randomly drawn tasks, one submission from each agent): mean normalized score (0–100) under the GPT-5.5 judge used throughout and under GPT-5.4. Agent ordering Spearman .83; task difficulty ordering Spearman .94; per-submission correlation .88.

LLM	GPT-5.5 judge	GPT-5.4 judge
GPT-5.5	70.3	72.5
Claude Opus 4.8	60.8	59.7
Doubao Seed 2.1 pro	72.6	74.8
MiniMax M3	56.9	53.4
Qwen3.7-max	57.5	53.8
DeepSeek-V4 Pro Preview	58.0	51.9

6 RELATED WORK

Repository-level evaluation. SWE-bench (Jimenez et al., 2024), its verified and multilingual extensions (Chowdhury et al., 2024; Khandpur et al., 2025; Zan et al., 2025), SWE-bench Pro (Deng et al., 2025), and feature-implementation benchmarks (Li et al., 2025; Zhou et al., 2026) evaluate executable changes. Other suites study freelance work (Miserendino et al., 2025), long-horizon development (Desai et al., 2026), upgrades (Xu et al., 2026), migrations (Hong et al., 2026), and tasks completed by running an existing repository or terminal environment (Ni et al., 2025; Merrill et al., 2026). SWE-CLIMB complements patch evaluation with explanation and operation interfaces, while retaining preservation tests in Evolution.

Context and measurement. RepoBench and CrossCodeEval study cross-file code completion (Liu et al., 2024b; Ding et al., 2023), while RepoQA, LongCoder, and LongCodeBench study long-context code understanding and completion (Liu et al., 2024a; Guo et al., 2023; Rando et al., 2025). Repository QA and documentation benchmarks evaluate answers and generated documentation for whole codebases (Peng et al., 2025; Cai et al., 2026; Chen et al., 2025; Hoang et al., 2025). Program-comprehension research studies how maintainers recover scattered facts (Ko et al., 2006; Sillito et al., 2006). Our atomic facts operationalize this need around a concrete requirement. SWE-agent, Agentless, and SWE-smith show the importance of scaffold and interface choices (Yang et al., 2024; Xia et al., 2025; Yang et al., 2025); we bind those choices to result versions rather than attributing all variation to the underlying model.

7 Discussion and Conclusion

As coding agents move into the systems the software world already runs on, the question that matters is whether they know those systems. SWE-Climb makes that knowledge observable at three rungs: explaining a change, operating the toolchain, and evolving the code without breaking it. Agent rankings differ between explanation and operation. Agents know how to change a system better than what must not change, and depend on the repository for the knowledge that matters most. By combining fact-level scoring with executable checks, SWE-Climb makes these maintenance capability gaps observable.

Limitations. All agents run in one shared Claude Code CLI scaffold, so each score reflects a model within that scaffold rather than within its native agent harness, and the Evolution rung covers four of the six agents.

AI USE STATEMENT

Generative AI is part of the evaluated coding-agent systems and the Exploration grading procedure described in Section 2.2. An AI assistant was also used to migrate the manuscript's LaTeX to the ICLR 2027 template, adjust figure placement, and check layout and internal consistency. The authors are responsible for the final text, claims, and artifacts.

ETHICS STATEMENT

The study concerns public software repositories and automated coding systems. Relevant release considerations include repository licenses, secrets and personal information in task artifacts, and the provenance and conditions of any human annotation. Tasks are distributed as pointers: upstream URLs, immutable revisions, and task overlays, rather than bundled third-party source, and each repository's license is recorded in the manifest.

REPRODUCIBILITY STATEMENT

Sections 2–4 describe track contracts, versioning, evaluation, and denominators. Appendix A specifies the canonical manifest fields, Appendix C the construction pipeline, Appendix D the grading contracts and contamination controls, and Appendix E the repository-access probe.

REFERENCES

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

Anthropic. Claude code settings: Attribution settings, 2026. URL https://code.claude.com/docs/en/settings#attribution-settings. Accessed 2026-08-27.

Songcheng Cai, Zhiheng Lyu, Yuansheng Ni, Xiangchao Chen, Baichuan Zhou, Shenzhe Zhu, Yi Lu, Haozhe Wang, Chi Ruan, Benjamin Schneider, Weixu Zhang, Xiang Li, Andy Zheng, Yuyu Zhang, Ping Nie, and Wenhu Chen. SWE-QA-Pro: A representative benchmark and scalable training recipe for repository-level code understanding. *arXiv preprint arXiv:2603.16124*, 2026.

Jialiang Chen, Kaifa Zhao, Jie Liu, Chao Peng, Jierui Liu, Hang Zhu, Pengfei Gao, Ping Yang, and Shuiguang Deng. CoReQA: Uncovering potentials of language models in code repository question answering. *arXiv preprint arXiv:2501.03447*, 2025.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Updated February 24, 2025.

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? *CoRR*, abs/2509.16941, 2025. doi: 10.48550/arXiv.2509.16941. 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.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems 36*, 2023. URL https://papers.nips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html.

GitHub. GitHub Copilot: The agent awakens, February 2025. URL https://github.blog/news-insights/product-news/github-copilot-the-agent-awakens/.

GitHub. Filtering and searching issues and pull requests, 2026. URL https://docs.github.com/en/issues/tracking-your-work-with-issues/using-issues/filtering-and-searching-issues-and-pull-requests. Accessed 2026-08-27.

Google. Gemini CLI: Your open-source AI agent, June 2025a. URL https://blog.google/innovation-and-ai/technology/developers-tools/introducing-gemini-cli-open-source-ai-agent/.

Google. Jules: An asynchronous coding agent, May 2025b. URL https://blog.google/innovation-and-ai/models-and-research/google-labs/jules/.

Maria I. Gorinova, Macey Baker, Amy Heineike, Maksim Shaposhnikov, Rob Willoughby, and Dru Knox. Position: Coding benchmarks are misaligned with agentic software engineering. *arXiv preprint arXiv:2606.17799*, 2026.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. LongCoder: A long-range pre-trained language model for code completion. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 12098–12107. PMLR, 2023. URL https://proceedings.mlr.press/v202/guo23j.html.

Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi D. Q. Bui. CodeWiki: Evaluating AI's ability to generate holistic documentation for large-scale codebases. *arXiv preprint arXiv:2510.24428*, 2025.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. SWE Refactor Bench: Can coding agents complete a long-horizon, whole-repository stack migration? *arXiv preprint arXiv:2608.23564*, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. SWE-bench multilingual, May 2025. URL https://www.swebench.com/multilingual.html. Official benchmark documentation.

Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. doi: 10.1109/TSE.2006.116. URL https://doi.org/10.1109/TSE.2006.116.

M. 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. URL https://doi.org/10.1109/PROC.1980.11805.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 17160–17176. Association for Computational Linguistics, July 2025. doi: 10.18653/v1/2025.acl-long.839. URL https://aclanthology.org/2025.acl-long.839/.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. RepoQA: Evaluating long context code understanding. In *First Workshop on Long-Context Foundation Models at ICML 2024*, 2024a. URL https://openreview.net/forum?id=hK9YSrFuGf.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=pPjZIOuQuF.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 44412–44450. PMLR, 2025. URL https://proceedings.mlr.press/v267/miserendino25a.html.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, Sen Hu, Ronghao Chen, Bo Li, Xin Li, Chen Hu, Binxing Jiao, Daxin Jiang, and Pin Lyu. GitTaskBench: A benchmark for code agents solving real-world tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*, 2025.

OpenAI. Introducing codex, May 2025. URL https://openai.com/index/introducing-codex/.

OpenAI. Separating signal from noise in coding evaluations, July 2026a. URL https://openai.com/index/separating-signal-from-noise-coding-evaluations/.

OpenAI. Why SWE-bench verified no longer measures frontier coding capabilities, February 2026b. URL https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions? *arXiv preprint arXiv:2509.14635*, 2025.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. LongCodeBench: Evaluating coding LLMs at 1M context windows. *arXiv preprint arXiv:2505.07897*, 2025.

Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 23–34. ACM, 2006. doi: 10.1145/1181775.1181779. URL https://doi.org/10.1145/1181775.1181779.

Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020. ISBN 9781492082798. URL https://abseil.io/resources/swe-book.

Hyrum Wright. Hyrum's law: An observation on software engineering, n.d. URL https://www.hyrumslaw.com/. Accessed 2026-08-24.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Xinbo Xu, Ruihan Yang, Haiyang Shen, Wendong Xu, Bofei Gao, Ruoyu Wu, Kean Shi, Weichu Xie, Xuanzhong Chen, Ming Wu, Jason Zeng, Michael Heinrich, Elvis Zhang, Liang Chen, Kuan Li, and Baobao Chang. RoadmapBench: Evaluating long-horizon agentic software development across version upgrades. *arXiv preprint arXiv:2605.15846*, 2026.

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*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601. URL https://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems 38*, 2025. URL https://papers.nips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems 38*, 2025. URL https://papers.nips.cc/paper_files/paper/2025/hash/5afa9cb1e917b898ad418216dc726fbd-Abstract-Datasets_and_Benchmarks_Track.html.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026.

A Asset Ledger and Reporting Protocol

This appendix records the release assets and the reporting rules that bind every number in the paper to a versioned manifest.

A.1 Asset facts

Table 5: Asset facts used in this paper.

Item	Value	Note
Fundamental task inventory	240	120 Exploration + 120 Operation
Repository identities / canonical URLs	131 / 130	one upstream migration alias explains the difference; both accountings ship in the manifest
Primary languages	20	project-level accounting
Tasks per repository	25 / 103 / 3	repositories contributing 1 / 2 / 3 tasks; none exceeds 3
Exploration gold facts	2,404	median 18 per task
Evolution tasks	57	three runs per agent

A.2 Canonical manifest minimum fields

The statistical unit of every reported number is a case row in a dated canonical manifest binding: sample ID, track, repo URL + SHA + snapshot date + archive hash; prompt/fixture/rubric-or-checker/hidden-test/scorer image hashes; dataset and exercise version, evaluation instance, case UUID, execution time; provider, model version, harness commit, system prompt, tools, budgets, sampling configuration, repetition index; raw grader state, functional score, invalid/error classification, token ledgers, and artifact paths; and the lineage of any content change, rerun, or exclusion.

A.3 Mutually exclusive result states

Operation and Evolution top-level states: `resolved`, `functional_failure`, `no_answer`, `answer_not_executed`, `gate_rejected`, `scorer_error`, `environment_error`, `timeout`. Only a behavioral failure after the checker executed the answer scores a benchmark 0. Exploration keeps per-item rubric scores with separate judge-error and missing-answer states and no binary label.

A.4 Reporting rules

Every reported number satisfies the following: (1) the caption's snapshot, denominator, repetitions, and error policy match the generating script; (2) case-level outputs and aggregates rebuild from one commit and agree; (3) results from different task versions are never spliced together; (4) creator, task-agent, judge, and provider token ledgers stay separate, unknowns stay NA; (5) conditioned analyses state their selection conditions and are never extrapolated to global rates; (6) Evolution results are never pooled with the fundamental tracks; (7) numbers agree across the paper, artifact card, dataset versions, and release repository.

B Worked Task Examples

Exploration. A MongoDB task asks for a tutorial on how the server keeps retryable-write state consistent when a replica-set member returns from a diverged history. All mechanism-locating terms (rollback, oplog, `config.transactions`, stable timestamp, ticket IDs) are removed from the prompt; top-weight rubric items require naming a private recovery hook and reconstructing a field-level query predicate that live in one 70KB C++ file among 2,783 files in the snapshot. General distributed-systems knowledge cannot recover these facts; reading the code can.

Operation. A Cypress task asks for a script that reports how the test runner resolves the effective configuration of several evaluator-provided projects, including values passed on the command line. The deciding fact is a precedence rule in the server source: a command-line key that belongs to one testing type is moved into that type's configuration block before the configuration file is merged, so it must override the file's value. Of 6,537 files in the snapshot (534k LOC), two source files establish the rule. All six agents fail the task in both runs: every answer lets the file's `baseUrl` win over the command-line value, while the reference and a structurally different equivalent answer pass.

C Construction Pipeline Details

Repository funnel. The fundamental tracks use a 50k core-LOC floor, and Evolution uses million-line-scale repositories; release manifests record the criteria version per task.

Per-sample pipeline. Every sample starts from traceable real evidence (issue, PR, comment, test, documentation, or source). Builders freeze the agent-visible snapshot; record upstream URL, commit, snapshot date, license, and network policy; then strip `.git`, issue/PR links, the final diff, hidden evaluation assets, credentials, and irrelevant caches. Prompts keep only legitimate user context—no target PR identifiers, future symbol names, or hidden file paths—and a de-identification gate requires locating keywords to grep to zero against the visible snapshot. Every content fix produces a new version and invalidates downstream results of older versions.

Admission gates. Exploration's core gate is recoverability: blind reviewers who never see the upstream implementation confirm each high-weight fact's atomicity, evidence, acceptable variants, and scoring boundaries against the visible base. Operation's core gate is production-contract parity: reference passes, wrong/empty fails, structurally different equivalent answers pass, every checker assertion is traceable to the prompt, and logs prove the agent's answer was actually executed. Evolution gates run in strict order—environment qualification (offline build, full test run within budget) before three-state closure before selection metrics before any expensive annotation—because inverting this order sinks annotation cost into tasks that can never be reproduced offline.

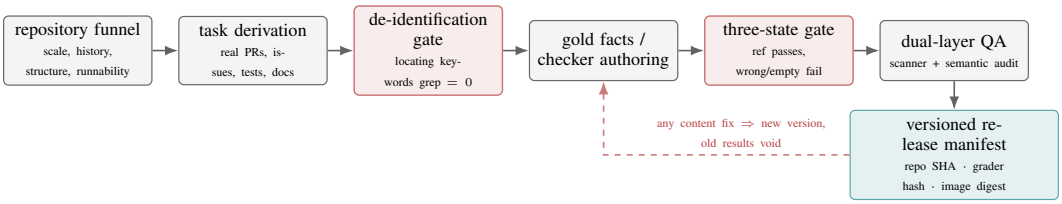


Figure 5: Release chain. Environment qualification and cheap gates run before expensive annotation; every artifact is bound to a dated canonical manifest, and content fixes invalidate downstream results of older versions.

D Grading Contracts and Contamination Controls

Deterministic Exploration aggregation. The judge returns item-level scores with quoted evidence; the grader recomputes every total from the items, applies clamping and capped hallucination penalties, and normalizes by the task maximum. A judge-reported total is logged for auditing and never enters a score.

Unified Operation checker contract. Every Operation checker implements one contract: it executes the submitted answer with the repository's own toolchain, asserts only behavior the prompt states, and reports process violations through a separate `gate_rejected` state, so that a contract rejection is never confused with a functional failure.

Evolution test admission. Hidden tests must not encode the gold patch's implementation choices: class names, package paths, method signatures, or error phrasings that the prompt does not disclose. Construction runs an automated gold-leakage check plus a manual sweep of reflection and by-name lookups in hidden tests, and uses a *mutant-first* ordering: cheap mutation probes establish that a candidate feature admits genuinely distinct wrong implementations *before* any full scorer is built. Admission targets tasks where the strongest current agent lands strictly between floor and ceiling.

Contamination hygiene. Evaluation disables networking; Git history and remotes past the base are stripped; PR merge dates are recorded against each model's training cutoff for stratified reporting. The task *forms*—writing a tutorial, reconstructing a run chain—have no public (input, answer) pairs to memorize, and the de-identification gate removes keyword shortcuts to the upstream change. A no-repo probe (prompt only, no repository) tests whether models can recite target files or symbols from memory; Evolution additionally runs a restricted-context shortcut baseline and a greenfield-hollow control that detects F2P sets passable by a disconnected reimplementation.

E The No-Repository Probe

Protocol. Every frozen task package was re-issued with two harness hooks and one prompt change, and nothing else: a setup hook removes the repository from the agent workspace and hides the image's pristine checker copy before the agent starts, a cleanup hook restores both for the judge (24 Operation checkers pass the workspace repository to the checker and compare it against the pristine copy), and the instruction gains a four-line preface stating that the repository is absent so the agent does not stall. Package identity was verified zip-by-zip against the frozen release (the only differences are the two hooks, the preface, and a variant tag). Agents were run with the same harness, budgets, and two trials per task; the judge and checkers are unchanged. *Leakage control.* Because an agent can search the filesystem, a scan of all 2,400 probe trajectories identifies every run that reached a repository copy, and every number below uses only runs that never touched one. For Exploration the copy is deleted outright, since its judge never reads the repository.

Additional probe summaries. In the row order above, score gains from repository access are 43.4, 47.5, 43.0, 30.7, 18.6, and 21.8 (rounding of aggregates can differ from subtracting displayed values). Mean tool calls change from 15 to 4, 47 to 4, 23 to 4, 29 to 20, 50 to 27, and 33 to 19.

Table 6: No-repository probe, clean runs only.

Track	Agent	With repo	No repo
Exploration	GPT-5.5	69.8	26.4
	Doubao 2.1 pro	69.9	22.3
	DeepSeek-V4	56.5	13.5
Operation	GPT-5.5	40.8	10.1
	Doubao 2.1 pro	25.2	6.6
	DeepSeek-V4	25.2	3.4

Exploration. Without the repository the fact-type ordering of §5.1 inverts: fix-path recall drops to 33 / 30 / 22 (GPT-5.5 / Doubao / DeepSeek) while legacy-invariant recall stays at 39 / 36 / 27 and validation-fact recall at 50 / 45 / 35. General knowledge of a project covers what must not change and how to test it far better than where the change lives, which is the fact class the repository uniquely supplies. The score difference measures sensitivity to repository access: Doubao 2.1 pro gains the most from the repository (48 points), so its near-parity with GPT-5.5 on Exploration is earned by reading the system, not by writing more without it. The tasks with the highest no-repository scores are dominated by well-documented upstream features (composite primary keys in Django, low-impact monitoring in CPython, match-label-keys in Kubernetes, Go toolchain installation); even there the with-repository score is higher in all but a handful of cases, and no task reaches the with-repository score of its best agent.

Operation and the no-repository screen. The probe doubles as a construction screen. An Operation deliverable that is a tool-generic audit, lint, or matrix script parameterized by the repository path (CMake presets, Helm rendering, YAML linting, `go vet`) can be written by an agent that knows the toolchain but has never read the system. Every task that two or more agents resolved blind was therefore revised to require a repository-specific entry point, precedence rule, legacy branch, or source-defined value, and re-validated with the three-state gate; the prompt states *what* must be handled and never *where*. After this screen, 67–85% of with-repository successes vanish without the repository. The eight tasks that two or more agents still resolve blind are in well-known projects whose repository-specific fact a strong model can reconstruct from memorized documentation; the release flags them (`norepo_resolvable`).

F Repository-Level Brownfield Profile

Table 7: Repository-level brownfield profile of the task cohort, computed by one scanner run at each repository's task snapshot (the construction snapshot commit recorded with the task where it is still reachable upstream, otherwise the last upstream commit before the recorded snapshot date or the construction cutoff), for all 130 canonical upstream repositories (one project has since been removed from GitHub and was scanned from its public archive mirror), and for the 54 upstream repositories behind the 57 Evolution tasks, each at the task's base commit (for one repository whose packaged base is not reachable upstream, the upstream base of the corresponding pull request). LOC are physical code lines counted by `scc`; staleness is the share of files tracked at the snapshot that no commit in the preceding five years touched.

Indicator	n	Median	IQR	Why it matters
Fundamental cohort (130 repositories)				
Snapshot LOC (code lines)	130	1.63M	843k–3.01M	the haystack the agent actually receives
Repository age (years)	130	13.6	10.5–17.7	integration time of "programming over time"
Commits in history	130	49.1k	24.6k–94.0k	accumulated change decisions; churn predicts defects
Distinct contributors	130	1,406	888–2,619	the Hyrum's-Law population; ownership dispersion
Share of files untouched $\geq$5 yr	130	9%	2%–17%	frozen expanses; knowledge evaporation
Evolution set (54 repositories, 57 tasks)				
Snapshot LOC (code lines)	54	1.34M	814k–2.27M	same haystack scale, one full feature or fix to land in it
Repository age (years)	54	11.3	8.9–15.4	younger on median, still a decade of accreted decisions
Commits in history	54	22.3k	10.9k–36.9k	fewer commits: newer or slower-moving projects
Distinct contributors	54	554	246–1,228	smaller ownership pool than the fundamental cohort
Share of files untouched $\geq$5 yr	54	3%	1%–14%	fewer frozen expanses; the tasks' difficulty is in the change, not in dead code

G The Ecosystem Signal: Measurement Protocol

Figure 6 deserves its measurement fine print. GitHub records neither "amount of legacy code" nor code quality, so we measure two narrow, observable proxies. *Declared debt*: the count of new public issues currently carrying a `technical-debt`/`tech-debt` label (hyphen and space variants), divided by all public issues in the same window—sampled bimonthly before GPT-4 and monthly/weekly after, without ever joining raw counts across resolutions. The normalized rate shows no jump after GPT-4; a BIC continuous-breakpoint search on the weekly series places the best break at 2025-10-27, months after the agent-CLI releases, and 77.5% of the July-2026 peak uses a single label variant—so the late rise mixes debt reporting with label adoption, and we treat it as a reporting signal, not an identified quality shift. *Agent adoption*: merged public PRs whose current body carries the default Claude Code attribution, a high-precision, unknown-recall proxy (Anthropic, 2026); attribution can be disabled or edited, search reflects the current body at crawl time, and multiple agent products launched in adjacent windows (GitHub, 2025; OpenAI, 2025; Google, 2025b;a), which is why the adoption curve names an ecosystem trend, not a product effect. Data: GitHub Search API snapshot, 2026-08-26 (GitHub, 2026).

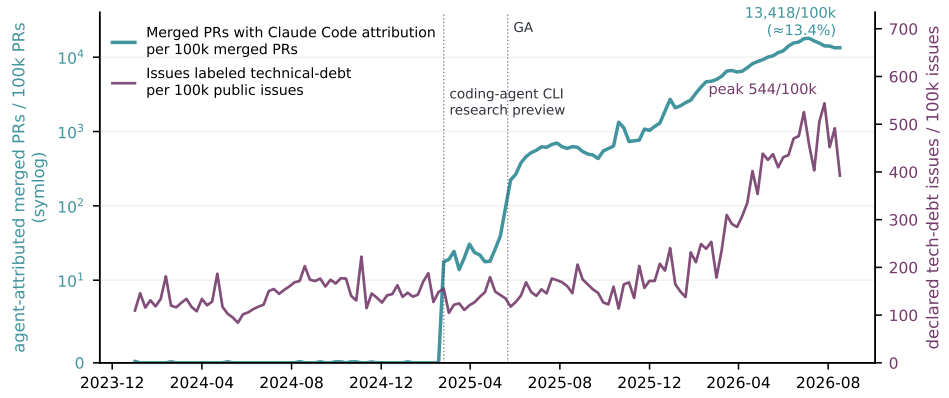


Figure 6: **Two descriptive ecosystem proxies.** Teal (left axis, symlog): merged public PRs whose current body carries the default Claude Code attribution, per 100k merged PRs (Anthropic, 2026), reaching 13,418 per 100k (≈13.4%) in the latest complete week. Violet (right axis): new public issues carrying an explicit `technical-debt`/`tech-debt` label per 100k public issues, peaking at 544 per 100k in July 2026. Both are shares of proxies with known biases (self-reported adoption; declared debt with a label-adoption confound); Appendix G gives the protocol and why we make no causal claim between the curves. Data: GitHub Search API snapshot, 2026-08-26 (GitHub, 2026).

H Mechanism Labels and Per-Agent Profiles

Table 8: Operation resolved rate by construction-time mechanism label (first two runs; task mean over the six agents; a task may carry several labels).

Mechanism	Tasks	Mean resolved
Platform / target matrix	9	15.7
Deep call chain	21	20.2
Behavior preservation (P2P)	8	21.9
Implicit compatibility / legacy path	27	23.5
Cross-module bridge	7	27.4
Build-system matrix	26	27.6
Scattered configuration sources	30	27.8
Configuration override layers	48	30.7
Generated-code layer	19	31.6
Monorepo navigation	16	32.3
Multi-stage pipeline	24	36.1
Registry / extension point	14	36.3
Hidden fixture semantics	33	38.4
Documentation gap or stale docs	7	40.5

Per-agent profiles. On Operation, Opus 4.8 has the highest resolved rate (40.4), ahead of GPT-5.5 (39.4; Table 1). GPT-5.5 has the highest Exploration score (69.4), with Doubao close behind (68.1) despite an Operation rate of 24.7. The remaining descriptors come from the first two runs: Opus 4.8 and GPT-5.5 use 25 and 29 tool calls per instance and uniquely resolve seven and six Operation tasks; Doubao has the highest invariant recall (68.4); MiniMax averages 86–102 tool calls per instance across the two tracks; Qwen has the lowest flip rate (15%); and DeepSeek has the largest number of authorized hallucination penalties (25).

I RUN-LEVEL STABILITY

Table 1 averages three runs per task. Mechanism profiles and the Operation difficulty distribution use the first two runs (Appendix H). Between those two runs, Operation outcomes agree on 79–85% of tasks, and Exploration's median absolute difference is 5–8 points.

I.1 TASK-RELEVANT FILE LOCALIZATION ANALYSIS

To further analyze the code-localization results in the main text, we use the first two runs of six models on Exploration tasks. For each run, we denote by R the number of distinct repository files accessed through successful dedicated `Read` calls, counting repeated visits to the same file only once. We denote by H the number of these distinct files that match parsable source/evidence paths recorded in the task's gold facts.

The association analysis in the main text compares two runs of the same model on the same task, yielding 707 pairs with complete item-level scores and mappable reference paths. For each pair, we compute changes in $\log(1 + R)$ and $\log(1 + H)$ and compare them with the change in normalized Exploration score. Holding the model and task fixed within each pair focuses this comparison on the relationship between changes in reading locations and changes in performance across two runs of the same task. The correlation between reference-file hits and score changes is nearly three times as large as that for the total number of files read, consistent with the main-text observation that task-relevant locations carry more performance information than simply expanding the reading scope.

I.2 REFERENCE-FILE COVERAGE AND ITEM-LEVEL SCORES

We further examine the relationship between reference-file coverage and item-level scores at the rubric-item level. For each Exploration item, we combine all parsable source/evidence paths in its associated gold facts and use the run's dedicated `Read` records to classify item-runs as all files hit, some but not all hit, or no observed hit; items without parsable reference paths are counted separately. An item-run is classified as all files hit only when its reference-path set is nonempty and every path is hit. Table 9 summarizes 27,338 item-runs with original item-level scores; full, partial, and zero credit are determined from item scores before run-level adjustments.

Table 9: Reference-file coverage and original item-level scores in Exploration. File coverage is computed from files observed in dedicated `Read` calls.

Reference-file coverage	Full	Partial	Zero	Total
All parsable files hit	7,185	4,485	704	12,374
Some but not all hit	2,179	3,538	649	6,366
No observed hit	1,297	3,544	2,648	7,489
No parsable reference paths	431	564	114	1,109
Total	11,092	12,131	4,115	27,338

Of the 12,374 item-runs that hit all parsable reference files, 4,485 received partial credit and 704 received zero, so 41.93% did not receive full credit. Thus, while file-level localization can indicate whether a model reached task-relevant source locations, it does not establish whether the code relationships that determine the answer were recovered and used in the final output. Below, we examine the code returned, subsequent analysis, and final deliverables in the three cases discussed in the main text.

I.3 TRAJECTORY EVIDENCE

Neovim. The task asks how view state remains consistent when a plugin or RPC modifies a buffer in a non-current split, including when the same buffer is displayed in multiple windows. GPT-5.5 primarily localized the issue to the current buffer (`curbuf`) and redraw-related logic, then read window-update code. In the returned code, the higher-level logic operates on the target window object `wp`, but its downstream helpers still depend on the global current window `curwin`, and some

branches also read the current window's height. The behavior of a non-current window therefore depends on the relationship between the target window and the current window's state.

Subsequent analysis remained focused on buffer access, and the final explanation likewise centered on the relationship between the modified buffer and the current buffer, omitting the dependency between the target and current windows. This case shows that task-relevant source code had been located and returned, but the cross-function state relationship governing behavior in the target scenario did not enter the final explanation.

CMake. The task asks why configuration-sensitive generator expressions on an imported interface target still fail to select the required dependency when `try-compile` explicitly uses the `Release` configuration and the project itself has a `Release` build type. Opus read the relevant configuration-handling logic and noted that the system distinguishes between an explicitly specified configuration and an empty configuration requiring a fallback.

Its main proposed fix, however, focused on the latter case: supplying a default configuration when the `try-compile` configuration is empty or unset. In the `Release` scenario explicitly given in the task, the empty-configuration fallback branch does not execute and the configuration remains `Release`, so the answer did not explain how the fix would change dependency selection in that scenario. The model also proposed a regression test for `Release`, but did not establish the connection between the test scenario and the main fix. This case shows that recovering local configuration rules is insufficient; proposed changes must also correspond to the conditions specified in the task under which they take effect.

Cypress. The task requires reconstructing the effective configuration with command-line overrides. In this e2e conflict example, the command-line `baseUrl` should override the value in the configuration file. Opus called the real underlying configuration merge function `mergeDefaults`, but did not reproduce the preceding argument processing: Cypress's actual entry point first moves command-line options specific to a testing type into the corresponding configuration block, then merges them with the configuration file.

Thus, although Opus executed the real merging logic, it supplied an input structure that differed from the native entry path, and the resulting `baseUrl` still came from the configuration file rather than the command line. Subsequent checks covered the output structure and other fields but did not verify this precedence rule. This case shows that calling a real underlying function does not by itself reconstruct the system's actual behavior; its upstream input transformations must also be traced, and the final output checked against the task requirements.

J Evolution Trajectory Evidence

J.1 Analysis setup

This section expands the evidence underlying the main-text Evolution trajectory analysis. For Superset, Presto, Angular, and Vert.x, we align the public behavioral requirements, source code returned by tools, the models' explicit interpretations of key mechanisms, final patches, and validation actually executed to examine behavior reconstruction and validation after localization.

J.2 Reconstructing behavior after localization

Superset. Notifications can target a subscribed user or guest identity (principal), or be narrowed to browser-tab (tab) routes. Two superficially empty states require different behavior: when no tab restriction is specified, principal-level fallback remains active; when an explicitly restricted tab set becomes empty after authorization filtering, delivery stops rather than widening back to the principal.

The no-restriction state is reachable through the subscription call chain. Task creation first subscribes the creator's principal, then invokes the tab-subscription policy; without a tab identifier, that policy records no tab consumer. Thus, with an existing principal subscriber, no recorded tab consumers, and an available message backend, delivery can reach the principal-level fallback. GPT-5.5 received the implementation encoding this fallback, but changed the policy to return the empty consumer list directly. The downstream manager treats that list as an explicit tab restriction, filters it to an empty route set, and stops delivery instead of taking the principal branch. The patch therefore collapses the distinction between an unspecified restriction and an explicitly narrowed set with no authorized targets.

The counterpart preserves `_consumers(task) or None` and explicitly distinguishes `requested is None` from a narrowed route set that becomes empty after authorization filtering. The former retains principal-level delivery; the latter stops delivery.

Presto. The same Parquet column can follow different paths under scalar (value-by-value) and batch reading. Reader dispatch sends scalar `FLOAT` values to `FloatColumnReader`, but batch reads to `Int32FlatBatchReader` or `Int32NestedBatchReader`. GPT-5.5 generalized a conversion property observed on the scalar path to the broader Parquet read pipeline. One solver explicitly inspected both scalar and batch dispatch, and its final patch handled `FLOAT`-to-`DOUBLE` conversion at the block-conversion stage. Its tests enumerated Parquet writer versions 1.0 and 2.0 with batch reading on and off, alongside ORC variants. The widening test executed six parameter instances with zero failures, errors, or skipped instances.

J.3 Closing the validation loop

GPT-5.5 on Vert.x. The trajectory identifies risks involving proxy TLS, origin TLS, nested TLS, and file transfer. The model continued to inspect relevant source near delivery, but did not con-

vert these recognized risks into executed behavioral checks. The visible validation ended with a successful build in which tests were skipped.

Angular. The task requires publishing declarations while an external dependency is absent, then consuming them downstream after that dependency becomes available. The test initially installed the dependency too early. One solver moved dependency installation to the downstream-compilation stage, so publication occurred without the dependency and downstream consumption occurred after installation. The corrected test returned 17 specs with zero failures, and the final patch preserved this publish–consume sequence. The final rerun reported 11 publishing specs and 17 downstream-consumption specs, both with zero failures.

The run also stashed the patch, reran the pre-change baseline in the same harness, restored the patch, and compared the failing and pending test-name sets. The extracted failing/pending test-name set was identical to the pre-change baseline. This baseline replay distinguishes failures already present in the pre-change baseline from regressions potentially introduced by the patch.

Vert.x. The final patch adds 17 proxy-TLS behavioral tests. The trajectory executed its 17 added behavioral tests, reporting 17 run with zero failures, errors, or skipped tests. These tests cover HTTP forwarding to a plaintext origin and HTTP/2 tunneling to a TLS origin, but omit HTTP/2 forwarding through an HTTPS proxy to a plaintext origin. This uncovered combination corresponds to the public requirements over connection type, origin TLS state, and proxy behavior.