BUILT ON BROKEN ASSUMPTIONS (BOBA): WHY CODING AGENTS CANNOT BE LEFT ALONE

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are autonomous developers. As such, they can help not only when each change is reviewed before the next is requested, but also when a whole sequence of requests is left to them, each built on the last. Although agents rarely notice their own mistakes and report wrong work as complete, evaluation has focused on a single request from human-written code, verified once. In this work, we build BOBA-BENCH, 280 chains of 20 consecutive requests from 140 real repositories, and compare agents on single requests and across chains. All five frontier agents we test complete far fewer chains than requests: 48% on average against 94–98%. Two matched comparisons show that frontier agents' verified code is as good a base as human code, and that the degradation comes from building on unverified code. Agents submit work as complete without confirming it and build on the broken state unaware, whereas one external "that's wrong" and a repair round lift chain completion by 8 to 16 points. In simpler terms, what limits coding agents left alone is not the code they write but the mistakes they cannot see: without verification, every request that follows is built on broken assumptions.

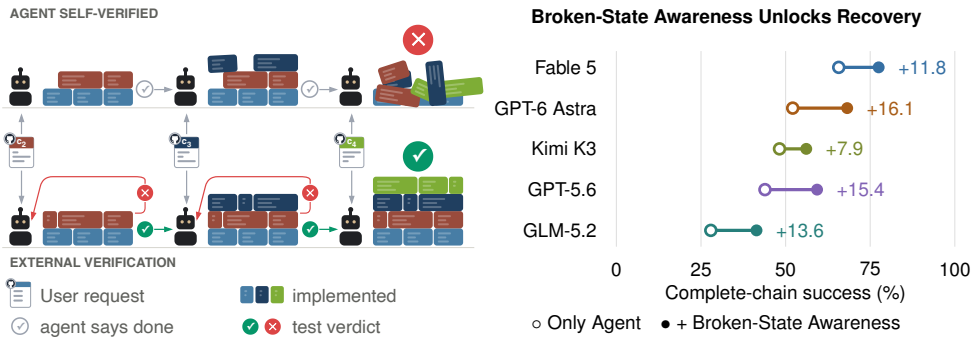


Figure 1: **An outside check lifts chain completion.** Left: unchecked work collapses; verification and repair preserve the foundation (schematic). Right: first-attempt completion of 280 chains without feedback versus "it doesn't work" and one repair attempt per failed submission (§5). Gains are in percentage points; Kimi K3's endpoint and gain are lower bounds (Appendix K.4).

1 INTRODUCTION

Today's coding agents function as autonomous developers (*e.g.*, Codex (OpenAI, 2025), Claude Code (Anthropic, 2025), OpenHands (Wang et al., 2025)), enabling developers to hand over a request and receive a finished change. Such delegation promises to help developers not only when they can review each change before asking for the next, but also when they cannot. In such cases, each request builds on code the agent wrote and reported complete (Liu et al., 2026a; Duma et al., 2026). Though trajectory studies have confirmed that agents rarely catch a mistake nothing flags for them (Rahman et al., 2026) and report wrong work as complete (Mehta, 2026; Advani, 2026; Smyth et al., 2026), coding agents are typically evaluated on a single request, from human-written code, verified once (Jimenez et al., 2024; Deng et al., 2025).

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.

Even though a growing body of work proposes to evaluate agents over successive requests (Deng et al., 2026; Lam et al., 2026; Orlanski et al., 2026; Chen et al., 2026; Shastry et al., 2026; Jin et al., 2026; Shen et al., 2026; Sobal et al., 2026), we identify in our review (Section 2) that prior work either carries an agent's verified and unverified work forward as one, even against an oracle control (Jin et al., 2026; Deng et al., 2026), so a later failure cannot be told apart as building on agent-written code or on a mistake nobody checked (Peng et al., 2026), or stops at the first failure (Shen et al., 2026; Sobal et al., 2026) and never continues from unverified work. We argue that unattended development, where mistakes go unnoticed (Rahman et al., 2026; Mehta, 2026; Advani, 2026; Smyth et al., 2026) and get built on (Rahman et al., 2026; Cuadron et al., 2025; Zhu et al., 2025; Laban et al., 2025), can only be measured by controlling what it lacks: verification.

In this work, we close this gap by creating a benchmark for continuous, unattended development – BOBA-BENCH – built from the commit histories of 140 real repositories. At a high level, the chaining process we propose transforms a stretch of a repository's history into a *chain* of 20 requests, specified from the tests the developers added, that jointly rebuild their changes. Autonomous evaluation then implements each request on the state the agent left behind, with hidden tests after every request telling verified work from a *silent miss*: a chain advances on verified work only and ends at the first miss, which we keep to resume from later.

On the 280 chains that we experimented on, we observed that agents left alone completed an average of 48% of chains, although the same agents pass 94–98% of the individual requests those chains are made of. Notably, we observe chains ending in a silent miss even within the first five requests, and across all five agents we test, from open-weights (Kimi K3) to state-of-the-art (Claude Fable 5).

Furthermore, we decompose the degradation into two components: (1) who wrote the code beneath a request, and (2) whether it was verified. We find that on verified code, more capable agents leave a foundation as buildable as the developers' own (*e.g.*, Claude Fable 5, GPT-6 Astra). On the other hand, all agents fail far more often on an unverified state, regardless of aptitude. We refer to this as the *built on broken assumptions* phenomenon: what limits coding agents left alone is not the code they write but the mistakes they cannot see, so that, without verification, every request that follows is built on broken assumptions.

We investigate several explanations for this effect and show that the agents tend to (1) submit work as complete without confirming it, leading them to (2) proceed onto a broken state unaware, (3) fail more as such mistakes accumulate, and (4) accept or reject finished work almost indiscriminately when asked to review it.

Finally, we test what the missing verification buys. Told only that verification failed, and given one repair round, agents fix a third to a half of their silent misses, and the repaired state is as sound as one that was never wrong, whereas restarting on the same state without the signal does not help. Repeated after every failed submission, this one message raises chain completion by 8 to 16 points for every agent (Figure 1), and a supervisor who is wrong one time in ten loses almost none of it.

Our findings highlight a gap between how coding agents are used in practice and how they are being evaluated. Ubiquitous silent misses are likely a reason why human verification remains the bottleneck (Krentsel et al., 2026; Winninger, 2026; Garousi, 2026), particularly for teams whose review of agent work is already thinning (Duma et al., 2026; Yu et al., 2026; Ye et al., 2026).

The rest of the paper is structured as follows: Section 2 reviews prior work. Section 3 describes the benchmark, its protocol, and the five agents we evaluate, and Section 4 presents and analyzes our main experiment. Finally, Section 5 tests external verification and draws recommendations.

2 BACKGROUND AND RELATED WORK

Early code-generation benchmarks asked models to implement isolated functions, checked against unit tests (Chen et al., 2021). As models began to edit real projects, repository-level benchmarks gave agents an issue and a human-written codebase and graded the resulting patch once (Jimenez et al., 2024; Deng et al., 2025). Later benchmarks enlarged the task to features, releases, and repositories (Zhou et al., 2026; Le et al., 2025; Zhao et al., 2025; Yang et al., 2026), but each task still began from a reference state and ended with a single verdict.

Table 1: BOBA-BENCH next to representative coding benchmarks. Bold: best among the repository-evolution benchmarks; underlined: second.

				Graded patches		95% CI	Per task		Task
Benchmark	Granularity	Tasks	Repos	per task	total	(±pp)	spec. words	patch lines	collection
Issue resolution									
SWE-bench Verified (OpenAI, 2024)	Commit	500	12	1	500	4.4	189	14	Rule-based
SWE-bench Pro (Deng et al., 2025)	Commit	1.9k	41	1	1.9k	2.3	462	170	Rule-based
CodeThread (Patel et al., 2026)	Commit	1.4k	78	1	1.4k	2.6	n/r	n/r	Rule-based
Codebase generation									
FeatureBench (Zhou et al., 2026)	Feature	200	24	1	200	6.9	4.8k	790	Test-driven
SWE-EVO (Le et al., 2025)	Release	48	7	1	48	13.6	2.4k	611	Rule-based
Commit0 (Zhao et al., 2025)	Project	54	54	1	54	12.9	>7k	>3k	Rule-based
DeepSWE (Huang et al., 2026)	Feature	113	91	1	113	9.1	302	701	Manual
Repository evolution									
SlopCodeBench (Orlanski et al., 2026)	Checkpoint	<u>36</u>	synth.	5.4	<u>196</u>	<u>15.5</u>	4.3k	n/r	Manual
SWE-Chain (Lam et al., 2026)	Release	12	<u>9</u>	12.9	155	24.6	<u>15.5k</u>	**8.9k**	Agentic
SWE-Milestone (Deng et al., 2026)	Milestone	7	7	<u>14.0</u>	98	29.8	**18.9k**	6.9k	Agentic
BOBA-BENCH (ours)	Request	**280**	**140**	**20**	**5.6k**	**5.8**	12.1k	<u>8.2k</u>	Agentic

Recent benchmarks instead evaluate agents over successive requests on the same codebase, so an agent's earlier implementation becomes the foundation for later work (Deng et al., 2026; Lam et al., 2026; Orlanski et al., 2026; Chen et al., 2026; Jin et al., 2026; Shen et al., 2026; Sobal et al., 2026). This exposes what single-request evaluation cannot: regressions, accumulated errors, and declining performance as development continues.

Crucially, these benchmarks differ in what happens after an agent fails. Some carry the failed work forward with everything else (Deng et al., 2026; Jin et al., 2026), so later failures mix the difficulty of building on agent-written code with the cost of building on a mistake nobody checked (Peng et al., 2026). Others stop at the first failure (Shen et al., 2026; Sobal et al., 2026), so the remaining evaluation concerns only verified code. Unattended development guarantees neither: agents can report wrong work as complete (Mehta, 2026) and proceed as though it were right (Rahman et al., 2026). We therefore make verification, not authorship, the central variable of our evaluation.

Separating the two requires controls that prior work provides only in part. CodeThread compares a follow-on issue on human-written and agent-written code (Patel et al., 2026), and EvoCode-Bench compares reference states with agent prefixes that passed cumulative verification (Shen et al., 2026). We run the same later request on human code, verified agent code, and unverified agent code, and then add repair: comparisons that separate what an agent leaves behind when right from when wrong.

Measuring these effects requires checking every request as the code evolves. Static measures describe the implementation (Paul et al., 2025; Sun et al., 2026), while completion of the current request ignores what earlier requests established. Like continuous benchmarks (Deng et al., 2026; Chen et al., 2026), we test new behavior alongside earlier behavior, with hidden tests from real development histories, so that an unresolved error, a later regression, and a repair stay distinguishable.

Scaling these experiments requires an external check that does not depend on the agent. Agents can debug from execution feedback (Chen et al., 2024) and write their own tests (Mündler et al., 2024), but their self-repair depends on the quality of that feedback (Olausson et al., 2024), and their tests can share their implementation errors (Tao et al., 2026). Model reviewers scale but remain imperfect judges (Wang et al., 2026), and human review does not scale (Krentsel et al., 2026). We use hidden tests as the check, ask whether agents can supply that check themselves, and measure how much external feedback is needed to recover. Appendix J provides supporting details.

3 BOBA-BENCH: BUILT ON BROKEN ASSUMPTIONS

To assess whether coding agents can be left alone, we develop a benchmark that repurposes the commit histories of real repositories. First, we apply a *chaining* process that turns a stretch of a repository's history into a *chain* of 20 requests, each specified from the tests the developers wrote.

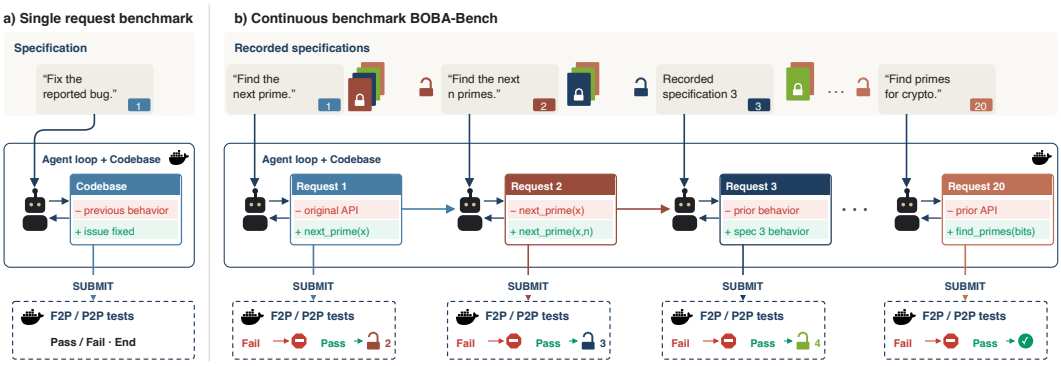


Figure 2: **Single-request versus autonomous evaluation.** (a) One specification, one submission, hidden F2P/P2P tests (SWE-bench shown). (b) BOBA-BENCH: 20 specifications arrive one at a time in one persistent codebase and conversation; after each submission only the git history reaches a fresh verifier, which runs the frozen contract of states $1..k$. A pass releases specification $k{+}1$; a failure ends the attempt.

Second, we implement an autonomous evaluation protocol that delivers the requests one at a time on the agent's own code, grades each against hidden tests, and tells the agent nothing about how it did. BOBA-BENCH contains 280 chains from 140 Python repositories, 5,600 requests in all (Table 1).

3.1 CHAINING PROCESS: FROM COMMIT HISTORIES TO CHAINS

Figure 2 contrasts a single-request task, one specification on human-written code graded once, with a chain. The chain is composed of a set of *requests*, each derived from a run of consecutive commits in the repository's history that change tests and source together, ending in one behavior change. More specifically, each request carries a specification, written from the tests the developers added rather than from their code, and a hidden *contract*: the added tests must fail before the change and pass after it, and the existing tests must keep passing. Taken jointly, the 20 requests of a chain rebuild the developers' changes in their order, on whatever code the agent has produced so far, and *state* k is the code after the first k of them.

As part of our work, we developed a semi-automatic chaining process to scale the construction of chains (Figure 3). First, requests are mined from the histories of Python repositories: wherever consecutive commits change tests and source together, the run is kept as a request only if its new tests fail before the developers' change and pass after it. Second, Claude Opus 4.8 writes each specification from the test diff, with commit messages and issue text as aids.

Because specifications are written by a model from tests the agent never sees, a chain could fail for reasons that are not the agent's. To understand this risk, we first attempted every specification blind, from the developers' code just before it: two models solve 78% of the released specifications between them, and the remainder were probed for behavior the tests demand but the text omits, then rewritten or kept as hard but fair. We then had three model families read every specification against its hidden tests, rewriting it until all three accepted identical text. Finally, each chain is sealed into one offline image whose dependencies are frozen for every commit, and released only if the developers' own patches pass every hidden test at every state in that image. We believe the process yields chains that are solvable from their text and on which a miss can be attributed to the agent, and we rely on it for our experiments; the details can be found in Appendix D.

3.2 AUTONOMOUS EVALUATION

Figure 2b depicts the process of evaluating an agent on a chain. At a high level, the loop involves two parties in two containers: the **agent** being evaluated, which works through the 20 specifications in one offline container and one conversation, and the **grader**, which runs in a fresh container and sees only the git history the agent commits.

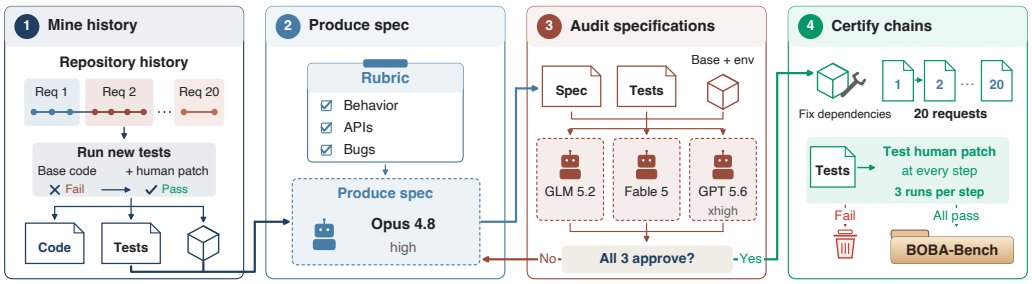


Figure 3: **Constructing BOBA-BENCH.** (1) *Mine history*: commits are cut into requests wherever tests and source change together, kept only if the new tests fail before and pass after the human patch. (2) *Specify*: Claude Opus 4.8 writes each specification from commit messages, issues and the test diff, not the source. (3) *Audit*: three model families must accept identical text. (4) *Certify*: 20 consecutive requests form a chain; human patches must pass all hidden tests in a frozen environment.

On the first request, the agent receives the codebase, the specification and a submit tool. It may edit and run code and tests, and submits once it believes the specification satisfied. The grader then applies the hidden test patches $1..k$ and runs the frozen contract of state k: the added tests of request k must fail before the agent's commit and pass after it, and the tests of every earlier request that the developers' own history still satisfies at k must pass. The chain ends if one of two conditions is met: (1) the agent has passed all 20 requests, or (2) a submission fails its contract, a *silent miss*, so called because the agent is never told: on a pass it simply receives the next specification, and on a miss the attempt ends.

Importantly, the agent is not told that it is being evaluated on a chain: withholding feedback is what unattended development looks like, so whether an agent can tell that its own work is correct is part of what is measured.

3.3 EVALUATION SETTINGS

We evaluate agents in five settings, each introduced here with its purpose.

AUTONOMOUS is the benchmark run described above: the agent is left alone, and a chain ends at its first silent miss.

AGENT CODE and HUMAN CODE isolate authorship. Request $k{+}1$ is solved on k requests that all passed their contracts, implemented either by the agent itself, with retries until each is certified, or by the developers, under identical specification, budget and grader; the comparison asks whether agent code is a worse base than human code once both are verified.

SILENT CONTINUATION isolates verification. A silent miss of the AUTONOMOUS setting is restored and request $k{+}1$ is issued as if the miss had passed, and we count failures of that request's own tests, not those inherited from the miss.

Table 2: Chain completion of five coding agents on the 280 chains of BOBA-BENCH. For each chain, completion is measured in two settings: pass@1, the first attempt completes all 20 requests, and pass@8, one of eight independent attempts does. reached@k is the share of first attempts that complete at least k requests, so that pass@1 is reached@20.

	Chains completed (%)		reached@k (%)			Requests completed	Silent-miss
Model	pass@1	pass@8	k=5	k=10	k=15	median (mean)	rate (%)
Claude Fable 5	65.7	79.6	90.7	81.8	75.0	20 (16.3)	1.9
GPT-6 Astra	52.1	61.1	90.4	74.6	62.9	20 (14.9)	3.1
Kimi K3	48.2	71.4	82.5	66.8	59.3	19 (13.8)	3.6
GPT-5.6-sol	43.9	62.1	87.5	70.0	57.1	17 (14.0)	3.8
GLM-5.2	27.9	47.1	72.5	55.0	40.0	11 (11.0)	5.7

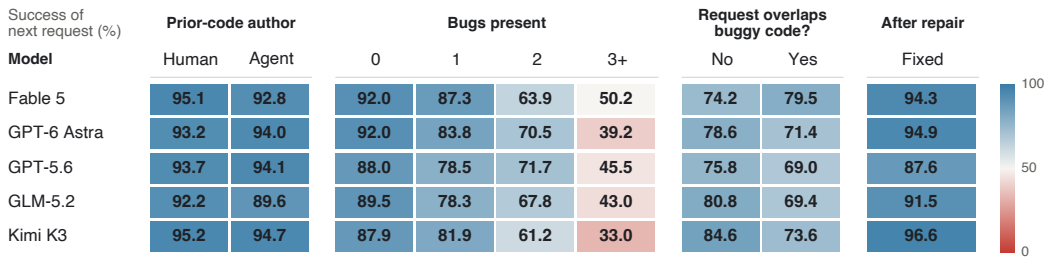


Figure 4: Success rate (%) of request $k{+}1$ for each model by the state it is built on: the author of the certified prior code (human history or the agent's own prefix), the number of unverified defects still open under the request (0, 1, 2, 3+), whether the request touches the code of the open defect (code overlap), and after one external "verification failed" message with one repair (Section 5).

VERIFIED takes SILENT CONTINUATION a step further: after the miss, a message from outside says that verification failed and the agent gets one repair before the next request.

REVIEW asks whether an agent can supply that message itself: a model judges a finished implementation without knowing its author, and accepts or rejects it.

3.4 SCALE AND PARAMETERS

We evaluate Claude Fable 5 (Anthropic, 2026), GPT-6 Astra (OpenAI, 2026b), GPT-5.6-sol (OpenAI, 2026a), Kimi K3 (Moonshot AI, 2026) and GLM-5.2 (Zhipu AI, 2026) on all 280 chains in the AUTONOMOUS setting, with up to eight attempts per chain, and then in the four other settings on the states those runs produced. As harness we use mini-swe-agent (Yang et al., 2024); each attempt may use unlimited model calls, 2 h of tool execution and 12 h of wall-clock time (settings in Appendix F).

4 CAN CODING AGENTS BE LEFT ALONE?

4.1 COMPLETION FINDINGS

Figure 1 and Table 2 summarize the results. At a high level, **every model sees its performance degrade when comparing the requests it passes and the chains it completes, with an average chain completion of 48%**. We name this phenomenon Built on Broken Assumptions: models that achieve stellar (94%+) performance on the individual requests they submit complete fewer than half of the 20-request chains made of *the exact same requests*, since a chain ends at its first silent miss.

In comparison, models perform only somewhat better when given eight attempts per chain, with pass@8 averaging 64%. This implies that the loss is not explained by bad luck in an attempt, which repetition would wash out. We observe that the same chains fail for every model (25% are completed by none of the five on the first attempt), and interpret this as indicating that difficulty resides in particular requests: a stronger model clears more of them, not different ones.

The silent-miss rate (Table 2) summarizes the magnitude of the Built on Broken Assumptions effect for each model. Surprisingly, **the miss rate does not fall as a chain progresses: more performant models (Claude Fable 5, GPT-6 Astra) build on broken assumptions at least as often at a chain's end as at its start, and so do weaker models (GLM-5.2)**, with per-request miss rates of 2-6%. This flatness is partly due to the protocol: in the AUTONOMOUS setting nothing compounds, so these figures show how soon the first unverified mistake arrives, not what follows it (§4.2, §5).

A different harness does not help models complete chains unattended, as the two models we re-ran under Codex CLI, the OpenHands SDK and pi fail in similar ways to their mini-swe-agent runs (Table 37). This result confirms that **the harness does not, on its own, allow models to verify their own work**. Nor is the effect explained by memorized code: only Claude Fable 5 shows memorization of these repositories, and no more of benchmark code than of other code (Appendix E).

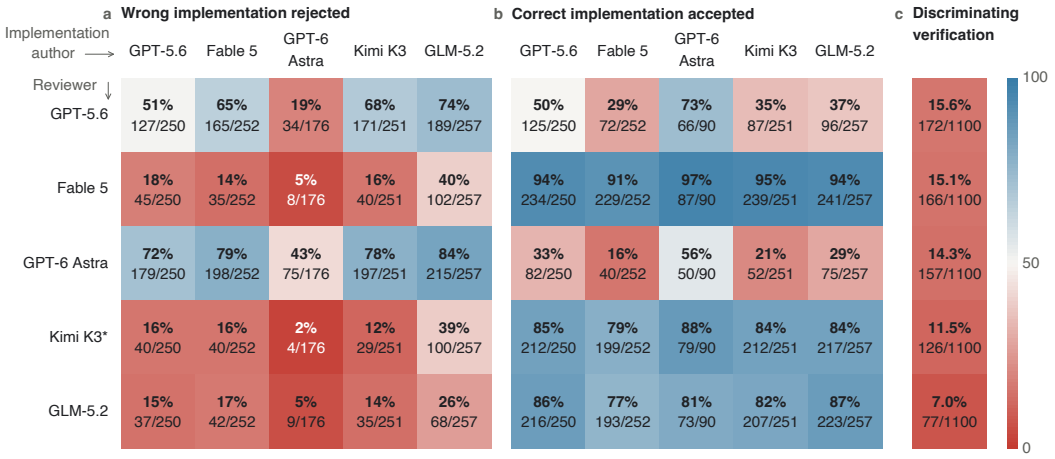


Figure 5: Rows: reviewers; columns: authors. Percentages of (a) wrong code rejected, (b) correct code accepted, and (c) matched pairs where reviewer tests fail wrong code and pass its correct twin.

4.2 Authorship vs. Verification Analysis

Table 2 concerns chains in which every request is built on verified code; we now separate who wrote that code from whether it was verified. Figure 4 visually summarizes the results. First, looking at authorship, we compare the Agent code and Human code settings (Appendix B). We see that models that build on their own verified code do as well as models that build on human code, with an average drop of 0.8 percent points. For instance, the four most able models succeed at the next request as often on either, at any depth. At the lower end, the least able model (GLM-5.2) is also the one whose own code is a measurably worse base than the human history. In summary, **when the code beneath a request is verified, models with higher aptitude leave code as buildable as a human's.**

The verification analysis paints a different picture. In the Silent continuation setting (Appendix F), failure of the next request's own tests rises sharply after a single unverified mistake, with an average increase of 18 percent points (nearly tripling). More interestingly, though better models tend to fail less often on a verified state, all models tend to pay a similar price for an unverified one. In other words, **on a state that nobody verified, all models we test exhibit very high failure, with the odds of failing roughly doubling with every additional unverified mistake beneath a request**. This refines our definition of the *built on broken assumptions* phenomenon: when comparing verified and unverified states, we find that large performance degradations are due in large part to mistakes going unverified, rather than to who wrote the code.

Appendix F.6 explores potential root causes for models building on broken assumptions. We identify four causes: (1) agents break behavior the request did not touch, the most common cause for every model (Appendix F.6.1), (2) they leave part of the specification unimplemented (Appendix F.6.2), (3) they implement the wrong logic or leave the new code unreachable (Appendix F.6.3), and (4) they trust their own tests, which barely lower the chance of a miss (Appendix F.6.4).

4.3 Cross-Model Review Experiment

The Autonomous setting leaves every request unreviewed, which may not reflect how agents are deployed in practice: a second agent could be asked to review each change, as prior work proposes (§2). We therefore ask whether an agent can keep models from building on broken assumptions.

In the cross-model review experiment, each of the five models reviews 2,286 items in the Review setting: every wrong implementation from the Autonomous runs, paired with a certified correct implementation of the same request by the same author (Appendix C).

Figure 5 summarizes the results. We find that all five models build on broken assumptions as reviewers: lenient reviewers (Claude Fable 5, GLM-5.2, Kimi K3) accept most wrong implementations,

strict reviewers (GPT-5.6, GPT-6 Astra) reject most correct ones. In other words, **any pipeline that relies on an agent to review an agent leads to models building on broken assumptions**.

5 TRUST, BUT VERIFY: WHAT AN OUTSIDE CHECK RECOVERS

Section 4 leaves a practical question: agents build on broken assumptions and cannot tell they have (§4.2, §4.3), yet nobody who deploys an agent has our hidden grader, and few will write a test for every request. Anyone who runs the result, however, can tell something is wrong without knowing what. **What is the least such a watcher must say for the agent to recover**, so that verifying costs as little as possible? We start from one bit, *verification failed*, delivered once after a silent miss with one repair round (the VERIFIED setting of §3.3), and ask whether repeating it, informing instead, or getting it wrong changes what it buys (Appendix K).

5.1 IMPLICATIONS FOR SYSTEM BUILDERS

The system around an agent has no test of its own, so it cannot judge a submission; only whoever wrote the request can. What it can do is keep a line open in both directions, so that the verifier sees the results and the model receives the verdict. Does one bit along that line suffice?

Told only that verification failed, **the models repair a third to a half of their misses in one round** (Table 33), and **a repaired state is no longer a broken assumption**: the next request fails on it no more often than on a certified state (Figure 4), whereas an unrepaired one fails like any silent miss (§4.2). The signal, not the fresh start, does this: restarting on the same state without the message does not help (Appendix K). Applied after every failed submission of a project, **completion rises by 8-16 points for every model** (Figure 1).

How much the verifier sees bounds how much it can say. For GPT-5.6 and Kimi K3, repeating the bare rejection after each failed repair helps with quickly diminishing returns, from 30 to 54-59% over eight rounds, with no round after the fourth adding more than 3 points and more than 40% of the misses remaining (Appendix K). **Naming the failing tests once achieves what insisting achieves in about three rounds**: insisting repeats the agent's belief about what it built; naming the test replaces it with a fact.

In short, system builders should keep that line open in both directions and widen what the verifier sees rather than multiply rounds: **more attempts are an inefficient substitute for telling the agent what is wrong.** Multi-agent systems are especially exposed, since models cannot judge a finished implementation (§4.3): the verifier's line must reach the subagents directly, not stop at the agent that delegated.

5.2 IMPLICATIONS FOR MODEL BUILDERS

Even a perfect supervisor recovers only part of the loss: more than half of the misses survive a repair round and 22-59% of projects still end short (Table 33), because the agent must find from one bit what it got wrong, cannot tell its wrong implementations from its correct ones (§4.3), and reports a wrong request complete a few times in a hundred (§4.1).

We therefore call on model builders to prioritize certifying a request as much as implementing it. **An agent that can be left alone should: (1) tell its wrong implementations from its correct ones, where the best model we test barely beats chance; (2) repair a miss in one round when told only that verification failed, where the models we test repair a third to a half; (3) achieve both without a supervisor, so that a chain does not depend on who is watching.**

5.3 IMPLICATIONS FOR BENCHMARK BUILDERS

What lets an evaluation observe agents building on broken assumptions is not the harness, which leaves completion within 9 points across four scaffolds (Table 37), **but three properties of the setting**: the agent's work persists across requests, correctness is decided by a contract the agent does not see and cannot rewrite, and the requests depend on one another. A single-request benchmark has none of the three, which is why the phenomenon has gone unmeasured.

We hypothesize that agents in any such setting will build on broken assumptions, and encourage benchmark builders to grade agents in chains, to report completion with and without a supervisor, and to publish the hidden contracts alongside the specifications, as we do.

5.4 Implications for Users

Users who delegate a sequence of requests should expect an agent to report a wrong request as complete and build on it. We make two practical recommendations.

Check before you build on it. If a request is the foundation of the next, run it or try the feature before issuing the next request rather than after: **a state built on a broken assumption fails the next request far more often (§4.2), whereas a repaired one is as reliable as one that was never wrong** (Figure 4). Checking after the fact is harder: by then the next request has already been built on the mistake.

Say what you saw. A wrong "wrong" costs little on its own: when we rejected correct submissions at random with probability 0.1 to 0.5 (GPT-5.6 and GPT-6 Astra, 100 chains; Appendix K), about 98% of the phantom repair rounds left the work passing. But alarms add up over twenty requests (Table 33): **rejecting one correct submission in ten costs at most 2 points of completion, and rejecting every other one gives back most of the gain**, though never below the AUTONOMOUS level.

These are patches, not a principled approach; once agents can certify their own work, users will be able to leave one alone with less risk of it building on broken assumptions, and recommendations like these will no longer be needed. Until then, the watcher's check is the one that works.

5.5 Release

Everything needed to reproduce and extend this work is available at `https://anonymous.4open.science/r/BOBA-bench-234E`: the benchmark and its construction pipeline, the data behind it (specifications, hidden contracts, gold patches and sealed images for every chain), the trajectories of every evaluated run, and the grader and analysis scripts behind every table and figure in this paper (Appendices D.12 and G). Images are pinned by digest and every artifact by hash, so results can be reproduced against the exact states we graded. Source code stays under each repository's license (Appendix A).

6 Limitations

BOBA-Bench measures fixed-test correctness, not user intent, readability or production readiness; it is Python-only, many repositories are small, and twenty requests do not imply a twenty-link dependency chain. The repositories come from public pools of agent tasks (Badertdinov et al., 2025; Fu et al., 2026), and nearly every change predates the model cutoffs, so models may have seen them; our memorization probe measures only near-verbatim recall (Fable 5 alone recalls 15.1% of sampled change blocks, others $\leq$1.2%, similar on same-repository controls; Appendix E).

7 Conclusion

In this work, we evaluate coding agents on twenty-request chains drawn from real repository histories, and find that on a fixed set of requests, agents that pass nearly all of them individually complete fewer than half of the chains they form. Coding agents build on broken assumptions, which materializes not as a loss of ability, since verified agent code is as good a base as human code, but as a failure of verification: a mistake the agent does not notice becomes the foundation of everything after it, and agents cannot tell right from wrong when asked. Additional experiments show that a single external "that's wrong" recovers much of the loss while more attempts and fresh starts do not, and we call on model builders to make certifying a request as much a goal as implementing it.

AI USE STATEMENT

Large language models are the subject of this study, and we also used them as tools; the ICLR 2027 AI Policy asks us to say how, so we go through its list. *Synthetic data and dataset cleaning (used).* Language models wrote every specification from the hidden test diff, commit messages, and linked issues and pull requests. Calls rotated over Claude Opus 4.6, 4.7, and 4.8, with GLM-5.2 as a fallback during outages; the model behind each specification was not recorded (Appendix D.6). Claude Opus 4.8 attempted each candidate specification blind, GPT-5.6 attempted those it did not solve, and Fable 5 and GPT-5.6 probed the rest for missing behavior (Appendix D.4). GLM-5.2, Fable 5, and GPT-5.6 audited each specification against its hidden tests, and GPT-5.6 and Fable 5 rewrote rejected ones (Appendix D.5). GPT-5.6 repaired failing environment builds by editing only their Dockerfiles, and sealing agents worked through a per-image checklist (Appendix D.7). GPT-5.6 assigned repository domain labels (Appendix A). We treat these outputs as generated data: the developers' hidden tests decide every grade, and every image must pass the 20 human patches at every state. *Qualitative and thematic analysis (used).* A judge model (GPT-5.6-sol) assigned one of 17 failure tags to each of the 575 first-attempt silent misses, and a second judge (Claude Fable 5) relabeled a stratified sample to measure agreement (Appendix F.6); a language model also labeled transcripts to check the self-verification rules of Appendix I. These labels are reported as indicative, with their agreement rates; every headline number comes from the hidden tests, not from a model's judgment. *Implementing methods and writing code (used).* AI coding assistants helped write and edit code under the authors' direction, including the agent scaffold, the grader, and the analysis scripts; the authors reviewed this code, and the grader is validated by requiring every sealed image to pass the 20 human patches at every state (Appendix D). *Methodology, hypotheses, and interpretation (used as assistants).* AI assistants provided feedback on experimental design and on the interpretation of results; the research questions, the experimental protocol, and the conclusions drawn are the authors'. *Not used or not applicable.* We did not use generative AI to develop theoretical models, formulate or prove mathematical claims, or translate text; the paper contains no proofs. Aside from the labeling above, no model judged any result. *Writing (used).* AI assistants helped with literature search, feedback on the manuscript, and drafting and editing of text, including figures and tables from the analysis outputs. The authors reviewed all AI-assisted work, checked every number against its source, and take full responsibility for the content of this paper, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

BOBA-Bench is built from the public histories of 140 open-source Python repositories; 128 of them use MIT (64), Apache-2.0 (35), BSD (17), or GPL-3.0 (12) licenses (Appendix A). Source code stays under each repository's license. No human subjects took part. Appendix D describes grader-defeat techniques found during construction so that benchmark builders can defend against them; the grader rejects them.

REPRODUCIBILITY STATEMENT

Section 3 and Appendix D describe construction end to end: mining and contract validation, the verbatim specification prompt (Appendix D.6), the specification audit, environments and sealing, grader rules, and every filtering round, including the post hoc 14-chain exclusion (Appendix D.10). Appendix A reports task statistics. Section 3.4 and Appendix F give models, settings, and budgets. Appendices B, C, and F.3 give the per-model estimates, and Appendix H gives the paired tests with their seed. Appendix G lists the population of each study. The 280 tasks with image digests, the grader, and the analysis scripts are released with the benchmark (Section 5.5).

REFERENCES

Laksh Advani. From confident closing to silent failure: Characterizing false success in LLM agents. *arXiv preprint arXiv:2606.09863*, June 2026. URL https://arxiv.org/abs/2606.09863. FAGEN Workshop at ICML 2026.

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, October 2024. URL https://arxiv.org/abs/2410.06992.

Anthropic. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet, February 2025. Announcement, February 24, 2025; documentation at https://code.claude.com/docs.

Anthropic. Claude Fable 5 and Claude Mythos 5. https://www.anthropic.com/news/claude-fable-5-mythos-5, June 2026. Announcement, June 9, 2026; system card at https://anthropic.com/claude-fable-5-mythos-5-system-card.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: an automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2505.20411. arXiv:2505.20411, May 2025.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-Agnostic SWE task collection at scale. In *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2602.23866. arXiv:2602.23866, Feb 2026.

Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. SWE-CI: evaluating agent capabilities in maintaining codebases via continuous integration. *arXiv preprint arXiv:2603.03823*, March 2026. URL https://arxiv.org/abs/2603.03823.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, July 2021. URL https://arxiv.org/abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to Self-Debug. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2304.05128. arXiv:2304.05128, Apr 2023.

Alejandro Cuadron, Pengfei Yu, Yang Liu, and Arpit Gupta. SABER: small actions, big errors – safeguarding mutating steps in LLM agents. *arXiv preprint arXiv:2512.07850*, November 2025. URL https://arxiv.org/abs/2512.07850.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. SWE-Milestone: evaluating AI agents on continuous software evolution. In *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2603.13428. arXiv:2603.13428, Mar 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*, September 2025. URL https://arxiv.org/abs/2509.16941.

Kacper Duma, Patryk Wróblewski, Jagoda Bobińska, Julia Winiarska, and Piotr Przymus. These aren't the reviews you're looking for: How humans review AI-Generated pull requests. *arXiv preprint arXiv:2605.02273*, May 2026. URL https://arxiv.org/abs/2605.02273. EASE 2026.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. daVinci-Env: open SWE environment synthesis at scale. *arXiv preprint arXiv:2603.13023*, March 2026. URL https://arxiv.org/abs/2603.13023.

Vahid Garousi. Human oversight and overload: Two hidden and costly burdens of AI-Assisted software engineering. *arXiv preprint arXiv:2606.05770*, June 2026. URL https://arxiv.org/abs/2606.05770.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: measuring frontier coding agents on original, Long-Horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, July 2026. URL https://arxiv.org/abs/2607.07946.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: can language models resolve Real-World GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770. arXiv:2310.06770, Oct 2023.

Qirui Jin, Lingching Tung, Kenan Li, Qiyang Shi, Yushi She, Huanzhong Jia, Harrison Zhao, Kejing Xia, Zhenbang Du, Jiaxin Pei, Zhenyu Zhang, Zhen Qi, Yuyan Duan, Wenke Lee, and Zijian Jin. ChainSWE: benchmarking coding agents on Multi-Bug software maintenance. *arXiv preprint arXiv:2607.02606*, July 2026. URL https://arxiv.org/abs/2607.02606.

Alexander Krentsel, Shubham Agarwal, Mert Cemri, Shu Liu, Sidharth Sankhe, Ziming Mao, Matei Zaharia, and Ion Stoica. Reality is the final verifier: On two key gaps in agentic software engineering. *arXiv preprint arXiv:2609.12039*, September 2026. URL https://arxiv.org/abs/2609.12039.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in Multi-Turn conversation. *arXiv preprint arXiv:2505.06120*, May 2025. URL https://arxiv.org/abs/2505.06120.

Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau sing Li, Jen tse Huang, Terry Yue Zhuo, and Michael R. Lyu. SWE-Chain: benchmarking coding agents on chained Release-Level package upgrades. *arXiv preprint arXiv:2605.14415*, May 2026. URL https://arxiv.org/abs/2605.14415.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: benchmarking coding agents in Long-Horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, December 2025. URL https://arxiv.org/abs/2512.18470.

Banruo Liu, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Ricardo Bianchini, and Esha Choukse. Agentic coding in the wild: Characterizing GitHub copilot traces at production scale. *arXiv preprint arXiv:2608.00101*, July 2026a. URL https://arxiv.org/abs/2608.00101.

Fengshuo Liu, Ying Liu, Ruize Sun, Lie Luo, and Siyuan Guo. Coding agents have converged: Why the SWE-bench leaderboard can no longer order its top entries, and what to measure instead. In *International Conference on Advanced Data Mining and Applications (ADMA)*, 2026b. URL https://arxiv.org/abs/2609.17394. arXiv:2609.17394, Sep 2026.

Aman Mehta. Confident and wrong: Silent semantic failures in coding agents. *arXiv preprint arXiv:2603.25764*, March 2026. URL https://arxiv.org/abs/2603.25764.

Moonshot AI. Kimi K3: Open Frontier Intelligence. https://github.com/MoonshotAI/Kimi-K3, July 2026. Model card and open weights, released July 16, 2026.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: testing and validating Real-World Bug-Fixes with code agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2406.12952. arXiv:2406.12952, Jun 2024.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is Self-Repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2306.09896. arXiv:2306.09896, Jun 2023.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/; dataset https://huggingface.co/datasets/princeton-nlp/SWE-bench_Verified, August 2024.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, May 2025. Announcement, May 16, 2025; cloud software-engineering agent, with the open-source Codex CLI at https://github.com/openai/codex.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, July 2026a. Released July 9, 2026; GPT-5.6 Sol is the flagship model of the family; system card at https://deploymentsafety.openai.com/gpt-5-6.

OpenAI. GPT-6 Astra: A new generation of intelligence. https://openai.com/index/gpt-6-astra/, September 2026b. Released September 3, 2026; system card at https://deploymentsafety.openai.com/gpt-6-astra.

OpenAI. Separating signal from noise in coding evaluations. https://openai.com/index/separating-signal-from-noise-coding-evaluations/, July 2026c. Audit of SWE-Bench Pro: $\approx 30\%$ of tasks judged broken, including underspecified and misleading prompts.

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

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: benchmarking how coding agents degrade over Long-Horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, March 2026. URL https://arxiv.org/abs/2603.24755.

Shaswat Patel, Betty Li Hou, Arun Purohit, Kai Xu, Jane Pan, He He, and Valerie Chen. Is agent code less maintainable than human code? *arXiv preprint arXiv:2606.21804*, June 2026. URL https://arxiv.org/abs/2606.21804.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. Investigating the smells of LLM generated code. *arXiv preprint arXiv:2510.03029*, October 2025. URL https://arxiv.org/abs/2510.03029.

Chao Peng, Zhiheng Lyu, Peijie Dong, Hande Dong, and Qiang Lin. Benchmarking the residual: What Long-Horizon evaluations add beyond matched Short-Task performance. *arXiv preprint arXiv:2607.27283*, July 2026. URL https://arxiv.org/abs/2607.27283.

Salman Rahman, Yubin Kim, Mihir Parmar, A. Ali Heydari, Genglin Liu, Simon A. Lee, Weizhi Zhang, Arian Hosseini, Ahmed A. Metwally, Yuzhe Yang, Baharan Mirzasoleiman, Xin Liu, Pavel Izmailov, Saadia Gabriel, Mark Malhotra, Shwetak Patel, Daniel McDuff, and Hamid Palangi. Locating hidden failures makes Long-Horizon agents more reliable. *arXiv preprint arXiv:2609.17930*, September 2026. URL https://arxiv.org/abs/2609.17930.

KN Ajay Shastry, Ganesh Senrayan, Shrey Satapara, Pranoy Panda, and Chaitanya Devaguptapu. Beyond isolated tasks: A framework for evaluating coding agents on sequential software evolution. *arXiv preprint arXiv:2604.03035*, April 2026. URL https://arxiv.org/abs/2604.03035.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. EvoCode-Bench: evaluating coding agents in Multi-Turn iterative interactions. *arXiv preprint arXiv:2605.24110*, May 2026. URL https://arxiv.org/abs/2605.24110.

Nolan Smyth, Yorguin-Jose Mantilla-Ramos, Pascal Jr Tikeng Notsawo, Saskia Helbling, Alberto Tosato, Mohamed Amine Merzouk, Nouha Dziri, Gauthier Gidel, and Tommaso Tosato. Quantifying overclaiming propensity in frontier LLM agents. *arXiv preprint arXiv:2609.20812*, September 2026. URL https://arxiv.org/abs/2609.20812.

Vlad Sobal, Shuo Yang, Yuting Zhang, Wei Xia, and Stefano Soatto. StaminaBench: Stress-Testing coding agents over 100 interaction turns. *arXiv preprint arXiv:2606.19613*, June 2026. URL https://arxiv.org/abs/2606.19613.

Xin Sun, Daniel Ståhl, Kristian Sandahl, and Christoph Kessler. Beyond resolved rate: A Non-Functional quality study. *arXiv preprint arXiv:2607.18462*, July 2026. URL https://arxiv.org/abs/2607.18462.

Leitian Tao, Baolin Peng, Haorui Wang, Hang Wang, Hao Cheng, Wenlin Yao, Qianhui Wu, Tao Ge, Sharon Li, and Jianfeng Gao. ExecCritic: learn to test, test to improve for coding agents. *arXiv preprint arXiv:2609.09133*, September 2026. URL https://arxiv.org/abs/2609.09133.

Ruoyu Wang, Jierun Chen, Shaowei Wang, Chaofan Tao, Sidi Yang, Yuxin Jiang, Kim-Hui Yap, Lifeng Shang, Xiaohui Li, and Haoli Bai. SWE-Review: closing the loop on issue resolution with agentic code review. *arXiv preprint arXiv:2607.06065*, July 2026. URL https://arxiv.org/abs/2607.06065.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741. arXiv:2407.16741, Jul 2024.

Thomas Winninger. Steerability via constraints: a substrate for scalable oversight of coding agents. *arXiv preprint arXiv:2607.02389*, July 2026. URL https://arxiv.org/abs/2607.02389. Deep Learning for Code Workshop at ICML 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 (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.15793. arXiv:2405.15793, May 2024.

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. Program-Bench: can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, May 2026. URL https://arxiv.org/abs/2605.03546.

Jingheng Ye, Huiqi Zou, Simon Yu, and Weiyan Shi. Coding with "enemy": Can human developers detect AI agent sabotage? *arXiv preprint arXiv:2606.05647*, June 2026. URL https://arxiv.org/abs/2606.05647.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: rigorous evaluation of coding agents on SWE-Bench. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. URL https://arxiv.org/abs/2506.09289. arXiv:2506.09289, Jun 2025.

Haoran Yu, Lifei Liu, Xiaochong Jiang, Yuwen Jia, Su Wang, Pin Qian, and Yihang Chen. Habituation at the gate: Rising approval and declining scrutiny in human review of AI agent code. *arXiv preprint arXiv:2606.22721*, June 2026. URL https://arxiv.org/abs/2606.22721. KDD 2026 Workshop on Agentic Software Engineering (SE 3.0).

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: library generation from scratch. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2412.01769. arXiv:2412.01769, Dec 2024.

Zhipu AI. GLM-5.2: Built for Long-Horizon Tasks. https://z.ai/blog/glm-5.2, June 2026. Released June 16, 2026.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: measuring LLMs' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270*, October 2025. URL https://arxiv.org/abs/2510.20270.

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. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2602.10975. arXiv:2602.10975, Feb 2026.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, September 2025. URL https://arxiv.org/abs/2509.25370.

A Task Statistics

All distributions in this appendix are over the 280 chains of Section 3 (5,600 graded states, 4,404 unique segments, 140 repositories); the unique-segment medians are within 3% of the occurrence medians for every size metric. Intervals are repository-clustered percentile bootstraps (2,000 replicates) for means and shares and Wilson 95% intervals where stated, because chains from one repository are not independent.

B Matched Foundations: Protocol and Controls

B.1 Per-model results

Table 9 gives the per-model estimates behind Section 4.2 on the 140 repositories: next-request success from the agent's certified prefix and from the human prefix over eight attempts per boundary, the agent-minus-human gap with 95% repository-bootstrap intervals (boundaries averaged within repository, then across repositories), the co-primary pass@8 gap, the gap by depth, the sensitivity that excludes boundaries whose next request already passed at k, and two descriptive counts: the share of boundaries where both prefixes pass all eight attempts, and the number of boundaries where the agent prefix passes at least four fewer attempts than the human prefix.

B.2 Ablation: earlier requests' tests visible in the repository versus hidden

Matched Foundations was run twice with one difference: whether the human tests of the already-accepted requests were present in the repositories the model worked in. In the first run, a constructed request was graded by applying its hidden test patch inside the agent's own container and the resulting tree was committed, so the certified agent prefix at boundary k carried the human tests of requests $1, \ldots, k$; the same patches were applied to the human prefix, so both arms saw them. For request $k{+}1$ these are its P2P tests (behavior that must keep working), never its F2P tests, which stayed hidden in both runs. In the second run, the one reported in the main text, the agent's tree is snapshotted before any hidden material enters the container and a throwaway copy is graded, so no hidden test appears in either prefix, matching the benchmark. Windows, depths, retry cap, scaffold and model settings are identical, so the boundaries complete in both runs form a paired ablation of P2P-test visibility (Table 10). Visible P2P tests lift both arms by 4–5 pp for the two models with clean runs in both conditions (GPT-5.6, Fable 5), but the agent-minus-human gap does not move: the differences are -0.4 to $+1.5$ pp with every interval spanning zero. The visible tests supplied regression protection and interface leakage (names, signatures and conventions the next contract assumes), and supplied them equally from either foundation. GLM-5.2's visible-tests construction ran through a proxy incident and Kimi K3's on a prefix-cached fleet that its hidden-tests run did not have, so their arm-level differences are not interpretable as a visibility effect; their gap differences are. GPT-6 Astra was run only with tests hidden.

C Cross-Verification: Protocol and Census

C.1 Per-model results

Table 11 gives the per-model estimates behind Section 4.3. The self-authored suites cover the four models in that study (one target per repository, eight suites each); sensitivity is estimable only on repositories where the model itself produced a wrong implementation. The review rows are equal-weight aggregates over the four original authors, with repository-clustered bootstrap 95% intervals.

D Benchmark Construction and Quality Assurance

This appendix expands the construction of Section 3.1 in the same order (mining, contracts, windows, specifications, audit, environments), then the protocol and grader of Section 3.2, and records the attrition from 25,519 verified segments to the 4,404 in the benchmark.

Table 3: Repositories (n=140). Size is measured at the base commit of the repository's first chain; stars queried 2026-09-09 (packaging 2026-08-27..30). Domain labels: GPT-5.6 from public metadata (fixed 13-way list; a keyword heuristic agrees on 53/140). Test style: pytest is the runner for every chain; style is inferred from the chains' own F2P test ids and test patches.

	p10	p25	median	p75	p90	max
GitHub stars	6	14	46	195	603	5,835
code LOC at first chain base	79	215	424	2,146	4,914	33,731
Python LOC	76	212	420	1,907	4,801	33,684
files	8	11	18	37	66	325
test LOC	9	29	97	538	1,317	11,574
chains per repository	1	2	2	2	3	5

Chains per repository: 34 repos with 1, 81 repos with 2, 18 repos with 3, 5 repos with 4, 2 repos with 5. **Primary language (GitHub):** Python 139, LookML 1; median Python share of code LOC at chain base 100% (12 repos below 90%). **Secondary languages present (repos):** Shell 15, JavaScript 10, CSS 7, HTML 5, C 2, TypeScript 2, C++ 1, Lua 1, SQL 1. **Domain:** api-client-sdk 21, language-parsing-analysis 17, scientific-numerical 15, general-utility 14, devtools-testing-packaging 13, data-engineering-formats 13, infra-cloud-ops 12, iot-hardware-home 10, web-backend-apps 9, media-games-graphics 6, text-docs-nlp 4, security-identity 4, finance-business 2. **Test style:** unittest.TestCase classes 63, pytest functions 60, pytest test classes 16, mixed 1; 51% of all F2P test ids are class-based. **Stars:** $<$10: 20, 10–49: 51, 50–99: 20, 100–499: 31, 500–999: 8, 1000–4999: 9, $\geq$5000: 1; 6 repositories archived; licences MIT 64, Apache-2.0 35, GPL-3.0 12, BSD 17.

Table 4: Per-segment statistics over the 5,600 segment occurrences (280 chains $\times$ 20). Mean is followed by its repository-clustered bootstrap 95% CI. Source commits: commits folded into the segment by the generation walk (see text). Contract nodes at k = F2P$_k$ $\cup$ P2P$_k$ $\cup$ still-valid earlier F2P.

	p10	p25	median	p75	p90	max	mean [95% CI]
Reference (gold) patch							
files changed	1	2	3	7	13	224	6.0 [5.3, 6.7]
hunks	2	4	7	15	30	1,654	15.5 [12.4, 19.3]
lines added	6	16	50	146	384	81,022	276.8 [179.8, 404.1]
lines removed	1	3	14	51	163	44,855	134.2 [83.8, 201.1]
lines changed (+/−)	9	24	73	215	596	94,155	411.1 [267.3, 579.1]
new files	0	0	0	1	2	135	1.0 [0.8, 1.2]
new functions (net new `def` names)	0	0	1	3	7	280	3.2 [2.7, 3.7]
new classes	0	0	0	1	2	469	1.3 [0.7, 2.5]
Hidden test patch and contract							
test files changed	1	1	1	2	4	73	2.5 [2.2, 2.8]
test lines added	4	12	31	79	178	24,704	109.0 [77.4, 165.4]
new test functions	0	0	1	3	8	160	3.3 [2.9, 3.7]
F2P tests	1	1	2	7	18	707	8.7 [6.8, 11.5]
P2P tests	7	22	45	91	208	1,303	93.3 [63.2, 136.3]
earlier F2P still demanded (accumulated)	2	9	23	45	87	835	41.9 [32.6, 54.6]
contract tests at k	14	29	51	100	216	1,312	102.0 [70.5, 145.2]
superseded earlier F2P (not demanded)	0	0	1	7	19	2,050	12.2 [5.8, 23.5]
Specification							
specification words	190	309	493	780	1,197	3,693	606.6 [577.1, 641.5]
spec words per test line added	4.0	7.2	14.6	32.2	80.9	1,419.0	36.2 [32.4, 40.3]
spec words per F2P test	33.9	80.4	178.0	352.6	598.0	3,068.0	267.8 [247.4, 289.4]
Source history							
source commits folded into the segment	1	1	3	6	12	163	5.2 [4.7, 5.8]
merge commits among them	0	0	0	1	2	35	0.8 [0.6, 0.9]
distinct authors (e-mail)	1	1	1	2	3	35	1.6 [1.5, 1.7]
segment duration, days (prev. target $\rightarrow$ target)	0.00	0.04	0.95	8.69	49.29	2,895.97	21.25 [16.63, 26.19]

Shares (Wilson 95%): P2P non-empty 96.0% [95.5, 96.5]; more than one source commit 68.9% [67.7, 70.1]; contains a merge commit 33.3%; more than one author 33.1%; creates a new file 28.7%; adds a function or class 62.4%; touches a non-Python file 56.0% (pure non-Python 0.20%); adds a new test file 23.9%; some earlier F2P superseded at k 64.6%; specification rewritten by a model during the spec audit 24.5%.

Table 5: Per-chain statistics (n=280 chains of 20 segments). Dependency depth = longest path in the DAG whose edges $j \to k$ ($j < k$) mark a segment whose gold or test patch references a symbol introduced, or imports a module created, by an earlier segment's gold patch (symbol/import definition), or whose gold patch touches a file the earlier gold patch touched (file definition).

	p10	p25	median	p75	p90	max	mean [95% CI]
gold lines added (20 segments)	781	1,246	2,177	4,274	7,013	121,416	5,536.6 [3,682.0, 8,007.4]
gold lines removed	278	442	775	1,836	3,396	56,611	2,684.6 [1,643.7, 4,033.9]
net gold lines	242	499	987	2,311	4,674	89,154	2,852.0 [1,552.4, 4,631.0]
distinct files touched by gold	11	16	25	45	74	297	38.0 [32.3, 44.7]
distinct test files touched	4	6	10	18	30	309	16.8 [13.2, 21.5]
new files created	2	4	10	24	46	201	19.1 [15.6, 23.4]
new functions	15	24	42	77	155	568	63.3 [53.2, 74.7]
test lines added	541	803	1,197	1,904	3,896	82,930	2,179.9 [1,546.1, 3,126.1]
F2P tests (sum over segments)	47	67	106	187	320	3,632	174.2 [137.7, 226.7]
contract tests at k=20	30	44	72	128	254	1,300	126.4 [93.2, 167.9]
specification words (sum)	7,794	9,583	11,502	13,696	16,518	34,430	12,132.9 [11,500.6, 12,804.8]
spec words per test line added	3.6	6.5	9.8	14.1	18.6	38.5	10.9 [9.8, 12.0]
spec words per F2P test	38.4	60.5	109.2	170.3	243.7	678.8	128.9 [114.7, 144.1]
segments with non-empty P2P (of 20)	18	19	20	20	20	20	19.2 [18.8, 19.5]
dependency depth, symbol/import (max 19)	2	4	6	8	11	17	6.2
dependency depth, file overlap (max 19)	14	16	18	19	19	19	17.0
code LOC at chain base	163	331	1,232	3,155	5,867	33,889	2,833.4 [1,980.3, 3,993.8]
files at chain base	9	14	28	50	85	336	42.4 [34.4, 53.5]
net gold lines / code LOC at base	0.07	0.24	0.90	4.61	10.31	125.97	4.62 [3.23, 6.21]
source commits folded into the chain	43	57	85	129	196	422	104.8 [94.5, 115.4]
commits before the chain base	1	4	40	116	200	529	80.6 [66.0, 96.7]
commit-date span, days	18	44	200	520	1,143	3,426	425.0 [336.0, 522.5]
median segment duration, days	0.05	0.19	0.99	4.54	14.97	77.48	5.09 [3.46, 6.96]
distinct human authors	1	2	4	9	15	39	6.5 [5.5, 7.5]

Shares: all 20 segments carry P2P 66.8% [61.1, 72.0]; single human author 14.6% [11.0, 19.3]. **Distinct human authors per chain:** 1: 41, 2: 47, 3–4: 62, 5–9: 72, $\geq$10: 58 chains. **Commit-date span:** $<$1m: 54, 1–3m: 46, 3–6m: 30, 6–12m: 46, 1–2y: 57, 2–4y: 29, $\geq$4y: 18 chains.

Table 6: Segment dependency structure over the 5,320 segment occurrences at positions 2–20. "Any earlier": some $j < k$ in the same chain satisfies the definition; "$k-1$": the immediately preceding segment does. CIs are repository-clustered bootstrap 95%.

definition	any earlier $j < k$	$k-1$	nearest lag (med.)	chain depth med. (IQR)
gold patches share a file	98.1% [97.2, 98.8]	89.2% [86.8, 91.5]	1	18 (16–19)
references a function/class touched by j	79.3% [75.5, 82.8]	40.1% [36.7, 43.5]	1	8 (5–11)
references a function/class introduced by j	72.3% [68.3, 76.2]	28.2% [25.6, 31.0]	2	6 (4–8)
strict (names ≥ 4 chars, non-generic)	68.2% [64.5, 71.9]	25.5% [23.1, 28.0]	2	5 (3–7)
gold-patch references only	64.6% [60.7, 68.6]	24.4% [21.8, 27.0]	2	5 (3–8)
test-patch references only	54.1% [49.8, 58.2]	15.0% [13.2, 16.9]	3	3 (2–5)
imports a module created by j	19.0% [15.8, 22.2]	4.0% [3.1, 4.9]	3	1 (0–2)
symbol introduced OR module import (headline)	73.3% [69.6, 77.0]	28.8% [26.1, 31.6]	2	6 (4–8)
file OR symbol OR import	98.9% [98.4, 99.3]	90.5% [88.2, 92.4]	1	18 (17–19)

Share of segments depending on the segment exactly ℓ positions earlier (symbol/import): ℓ=1: 29%, ℓ=2: 27%, ℓ=3: 24%, ℓ=5: 23%, ℓ=10: 21%, ℓ=15: 22%, ℓ=19: 27%. File overlap: ℓ=1: 89%, ℓ=2: 85%, ℓ=5: 78%, ℓ=10: 72%, ℓ=19: 66%.

D.1 SEGMENT MINING

Candidate repositories are Python projects for which SWE-rebench (Badertdinov et al., 2025; 2026) or OpenSWE (Fu et al., 2026) publish a working test image: 5,124 images over a pool of 10,319 repositories. For each repository the generator iterates the commits reachable from the default branch in oldest-first commit-date order (`git rev-list --reverse`), not a first-parent path, and groups consecutive commits into a *candidate segment* as soon as the accumulated change touches both test files and source files; files are classified by path, and no file belongs to both patches. The changed tests become the hidden *test patch*, everything else the *gold patch*. A candidate is verified by execution (§D.2); a candidate that fails verification is not discarded but absorbed into the next candidate (fold-forward), so segments are contiguous and the target commit of segment i is the base of segment $i{+}1$. No size limit is applied. The walk produced 25,519 verified segments across 1,491 repositories.

Because the walk orders commits by date rather than by integration, the base of a segment is an ancestor of its target for 92.3% of boundary pairs (Wilson 95% CI [91.6, 93.0]); the remaining 7.7% hop between branches, and the gold patch is in every case the tree diff between the two boundary states. Only 71.7% of boundaries are first-parent-reachable. The 4,404 unique segments of the benchmark fold 23,407 source commits in walk order (median 3, mean 5.3, maximum 163; 25,505 by `rev-list base..target`, 19,145 first-parent). Chains start at the first tested commit of a repository, so a median of about 40 commits precede the first chain base and are covered by no segment.

D.2 EXECUTABLE CONTRACTS

Verification runs in fresh containers: the full pytest suite at the base with the test patch applied, then again with the gold patch added, three runs in each state so that a flaky test cannot enter the contract. *Fail-to-pass* (F2P) nodes are tests in the changed test files that do not pass before and pass after; *pass-to-pass* (P2P) nodes pass in both states. Runner errors are separated and retried rather than counted, so a broken collection can never manufacture an F2P, and at least one F2P node is required. The two node lists are the segment's *contract*. Throughout the benchmark a state passes iff every listed node reports the literal outcome `passed`; skipped, xfail/xpass, errored and absent nodes all fail. Because the human patch has been shown to satisfy the contract in the same environment, a later failure is attributable to the submission, not to the tests or the environment.

D.3 FROM SEGMENTS TO CHAINS

Attrition before windowing. The 25,519 verified segments were processed as two repository-disjoint pools; 14,921 belong to the primary pool. A chain requires 20 consecutive verified segments in one repository, so only the 6,316 primary-pool segments (141 repositories) lying in continuity runs of at least 20 were candidates. Strict retained-behavior eligibility removed 1,178 of these, and 240 more fell with 22 repositories under the 20-run floor, leaving 4,898 segments in 119 repositories. Interior removals were made chain-safe: the nearest retained predecessor was extended to the next retained base (112 interior collapses, 65 boundary runs), and for each changed range the target metadata, source-only gold patch, hidden tests, exact F2P/P2P lists, Dockerfile metadata and specification were regenerated; every retained segment kept at least two passing F2P tests. All 4,898 pass the strict offline Oracle. To these were added 1,842 Oracle-verified segments from continuous runs of the second pool (drawn from its 8,368 Oracle-passing segments).

Windows and task materialization. Chains are cut only inside repository-local runs in which every consecutive segment passed the Oracle: non-overlapping windows of exactly 20, plus one final window aligned to the run's end when its length is not a multiple of 20; shorter fragments are discarded, never bridged. This gave 439 windows over 193 repositories (119 from the primary pool, 74 from the second; 6,740 unique segments, 8,780 occurrences) before filtering; specification unanimity (§D.5) left 397 windows over 176 repositories and rounds 4–15 (§D.10) left 280 over 140. A *task* is an immutable base commit; 20 ordered specifications, a later one taking precedence where it intentionally supersedes earlier behavior; 20 frozen hidden-test patches with exact F2P/P2P lists per state; 20 frozen source-only gold patches used only by the Oracle; and an immutable, digest-addressed environment. All 22,400 contract files in the benchmark are byte-identical to the freeze.

The overlapping final windows make 1,196 of the 5,600 segment occurrences repeat a segment of the previous window.

D.4 Specification authoring and solvability

One call to Claude Opus 4.8 (high reasoning) receives the segment's commit messages, linked issue and PR bodies and the full test diff, never the gold patch. It must treat the tests as the sole source of truth, name the public API they import, state behavior and edge cases rather than implementation, add nothing the tests do not cover, and never mention tests or assertions. The verbatim prompt and generating function are in §D.6.

Every released segment carries an offline *Oracle* proof: the gold target, checked out in a history-scrubbed, network-free container, passes its exact F2P/P2P lists.

Blind solvability campaign (Stage A, 7–17 August 2026). The 14,921 candidate specifications of the primary pool were attempted blind from the human prefix in a sequential design: Claude Opus 4.8 attempted every candidate eight times; GPT-5.6-sol (xhigh) then attempted eight times only those Opus never solved (Table 13). Specifications that both models failed were probed by Claude Fable 5 and GPT-5.6-sol for behavior the hidden tests demand but the text omits. Inside the material that could still become benchmark chains, every flagged specification (294) was rewritten and re-attempted blind four times by GPT-5.6-sol (75 solved; 219 validations were cut short by harness failures and remain `repaired_pending`); flagged specifications outside that material were not rewritten, since their segments left with their runs. Never-solved specifications with no gap found were kept as `hard_but_fair` on the strength of the strict offline Oracle and the two spec-gap verdicts. Non-solve was recorded as evidence and never treated as invalidity.

Composition of the benchmark. Selection favored solved specifications (37% of blind-solved candidates reached the benchmark against 17% of dual-failed ones), but one benchmark specification in five was never solved blind by either model in 16 attempts (Table 14). These are spread evenly over chain positions (47–67 per position) and touch 204 of the 280 chains (76 chains have none, the median chain has one or two, 9 chains consist entirely of never-solved specifications). They were retained on the strength of the strict offline Oracle and model-based adequacy adjudication: in the primary pool they carry the v1.2 statuses `hard_but_fair` (575), `repaired_validated` (45: specification rewritten, then solved in blind pass@4), `repaired_pending_validation` (66) and `repaired_pending_pass4` (2); the 1,305 second-pool segments have no status field but the same campaign classes apply (966 Opus-solved, 68 GPT-only, 205 dual-failed, 66 incomplete). specification as it stood in August 2026; the review of §D.5 later rewrote 22.7% of specifications.

D.5 Adequacy audit (Stage B, 21–30 August 2026)

Criterion. Does the specification give a competent engineer all *publicly relevant* behavior needed to satisfy the hidden tests (APIs, inputs and outputs, ordering, exceptions, edge cases, compatibility) without dictating internals?

Reviewers and sessions. Three reviewers from different model families (GLM-5.2, Claude Fable 5, GPT-5.6 at xhigh reasoning) judge each specification with the base code, hidden tests and F2P/P2P lists available and the gold patch withheld, with no token, message, tool or time limit (model-generated compaction only); provider errors are infrastructure, never votes. Because segment k presumes all before it, reviews run as sealed sequential sessions: the 20 base states are encrypted with per-segment keys, only segment k is decrypted, and a structured verdict (decision, missing contracts with evidence, contradictions, confidence) must be submitted before the key is destroyed and $k{+}1$ is revealed. A specification is accepted only when all three return SPECIFIC_ENOUGH for the same SHA-256 of its text.

Waves and freeze. The review ran after windowing, on the specifications inside the 439 candidate chains: 6,740 materialized, 6,716 after one abnormal segment was removed, 6,688 after two `google/gcp_scanner` chains were lost to a provider content filter (provider failures are never votes). In wave 1, 4,989 (74.6%) were unanimous at once; reviewer strictness differed (INSUFFICIENT rates 10.6% GLM-5.2, 12.0% Fable 5, 21.5% GPT-5.6). The 8,539 missing-contract findings

on the remaining 1,699 fed rewrites by GPT-5.6 (1,459 targets after source filtering, 1,389 accepted, 95.2%) and then by Claude Fable 5 (70 of 101), each re-entering fresh sealed review by all three reviewers, followed by a hygiene pass for leaked test language. The 62 still contested were discarded; because a discard splits its run and fragments shorter than 20 are dropped, 560 collateral segments and 15 repositories left with them, and the runs were re-windowed: 397 chains over 176 repositories with 6,066 specifications, of which 4,607 (75.9%) keep their original text, 1,389 (22.9%) are GPT-5.6 rewrites and 70 (1.2%) Fable 5 rewrites. Every specification in the benchmark is unanimously accepted, and a lint finds none of its 4,404 specifications mentioning hidden tests, graders, gold or node IDs. A specification can be *hard but fair* from §D.4 (never solved blind, adjudicated adequate) and unanimously accepted here: 575 benchmark specifications are exactly that (Table 15).

Disclosed limits. Authors and reviewers saw the hidden tests, so specification and tests are not independent; and no human-labeled adequacy sample exists yet.

D.6 SPECIFICATION-GENERATION PROMPT AND CODE

The specification of every segment is produced by one call of the function below (`_build_spec_incremental` in `pipeline/dind_worker.py` of the SWE-Build generator), sent as a single user message to a chat-completions endpoint at temperature 0.3 with `max_tokens` equal to the remaining context window, no tools and no repository access. Its inputs are the segment's commit messages, its linked GitHub issues and PRs, and the segment's test diff; the gold source patch is never an input. The proxy was configured for Claude Opus 4.8; the campaign record describes round-robin over Opus 4.6/4.7/4.8 with GLM-5.2 as a manual fallback during outages, and the per-segment author model was not recorded. Over-long lines of the function were re-wrapped for the page (adjacent string literals and parenthesized continuations; the parse tree is identical to the archived file). If the call overflows the context window it is repeated with the narrative blocks dropped and only the test diff kept. Both files are archived under `benchmark_construction/spec_generation/` in the release; the template's SHA-256 is `e667281d068ad30f2a698f9b4b7ac494e3a98d4656ac0e2fb491715f8345beea`.

Prompt template. Line breaks added for width and the em dash shown as `--` (the typewriter font has no em dash); `{context}` is filled as shown in the second listing.

```
You are writing a task specification for a software engineer who must
implement a change to a codebase. The TESTS ARE THE SOURCE OF TRUTH: the
spec must describe exactly the behavior the test changes below require --
nothing more, nothing less. Anything not exercised by the tests does not
exist for this task; do NOT invent requirements from the commit messages
or issues that the tests don't actually verify (treat those as weak
hints only, and ignore them where they disagree with the tests).

CRITICAL CONSTRAINT: the engineer will NOT be able to see the tests.
Your specification is the ONLY thing they get. Therefore, derived PURELY
from what the tests exercise:
- Describe the required behavior -- expected inputs, outputs, edge cases,
  error conditions -- for exactly what the tests check.
- NAME the public API surface the tests import and call: module paths,
  class names, function/method names and signatures (the engineer must
  create these so the hidden tests can import and call them).
- Do NOT reference the tests themselves: no test file/function names, no
  assertion snippets, no 'the test expects...'. Describe the BEHAVIOR
  the tests imply, not the tests.
- Do NOT describe the implementation -- state WHAT is required, not HOW.
- Do NOT add behavior the tests don't cover, even if commit
  messages/issues mention it.

{context}
Write the self-contained, test-grounded problem statement now:
```

Context blocks and overflow fallback.

```
--- {context} is built from these blocks, each included only when
non-empty ---
=== COMMIT MESSAGES (in order) ===
- <commit message 1>
- <commit message 2>
...
```

```
=== LINKED GITHUB ISSUES / PRs ===
Issue #<n>: <title>
<body>
PR #<m>: <title>
<body>
...

=== TEST CHANGES (hidden from the solver -- describe their behavior,
never reference them) ===
<full test diff of the segment>

--- On context overflow the call is repeated with the narrative dropped
and only this block ---
=== TEST CHANGES (source of truth -- describe the behavior they require,
never reference them) ===
<full test diff of the segment>
```

Generating function.

```
def _build_spec_incremental(per_commit_dir, github_issues_by_number, vllm_url,
                            overall_test_patch=""):
    """Build ONE self-contained problem statement per TESTABLE UNIT (segment).

    SWE-Build change: the unit of spec generation is the testable segment, not
    individual commits. We aggregate ALL of the segment's signal into a single
    LLM call:
      - every commit message in the segment
      - every linked GitHub issue / PR body
      - the segment's full test diff (the behavioral ground truth)

    The solver agent NEVER sees the tests, so the spec must translate test
    behavior into self-contained prose (name the public API surface the tests
    exercise, describe expected behavior/edge cases) WITHOUT referencing test
    files, test function names, or assertions.

    Signature preserved for the caller. Returns (list_of_specs, issue_generated).
    """
    # ---- aggregate commit messages ----
    messages = []
    if per_commit_dir.exists():
        for cdir in sorted(per_commit_dir.iterdir()):
            if not cdir.is_dir():
                continue
            mf = cdir / "message.txt"
            if mf.exists():
                m = mf.read_text().strip()
                if m:
                    messages.append(m)
    # ---- aggregate linked issues/PRs (dedup by number) ----
    issue_text = ""
    issue_used = False
    seen = set()
    if per_commit_dir.exists():
        for cdir in sorted(per_commit_dir.iterdir()):
            rf = cdir / "issue_refs.txt"
            if not (cdir.is_dir() and rf.exists()):
                continue
            for ref in rf.read_text().strip().split("\n"):
                ref = ref.strip()
                if not ref or ref in seen:
                    continue
                try:
                    num = int(ref)
                except ValueError:
                    continue
                seen.add(ref)
                iss = github_issues_by_number.get(num)
                if iss:
                    label = "Issue" if iss.get("is_issue") else "PR"
                    issue_text += (f"\n{label} #{num}: {iss.get('title','')}\n"
                                   f"{iss.get('body','')}\n")
                    issue_used = True

    if not (messages or issue_text
            or (overall_test_patch and overall_test_patch.strip())):
        return [], False

    context = ""
    if messages:
        context += ("=== COMMIT MESSAGES (in order) ===\n"
```

```
                + "\n".join(f"- {m}" for m in messages) + "\n\n")
    if issue_text:
        context += "=== LINKED GITHUB ISSUES / PRs ===" + issue_text + "\n"
    if overall_test_patch and overall_test_patch.strip():
        context += ("=== TEST CHANGES (hidden from the solver -- describe their behavior, "
                    "never reference them) ===\n" + overall_test_patch + "\n")

    prompt = (
        "You are writing a task specification for a software engineer who must implement "
        "a "
        "change to a codebase. The TESTS ARE THE SOURCE OF TRUTH: the spec must describe "
        "exactly "
        "the behavior the test changes below require -- nothing more, nothing less. "
        "Anything not "
        "exercised by the tests does not exist for this task; do NOT invent requirements "
        "from the "
        "commit messages or issues that the tests don't actually verify (treat those as "
        "weak hints "
        "only, and ignore them where they disagree with the tests).\n\n"
        "CRITICAL CONSTRAINT: the engineer will NOT be able to see the tests. Your "
        "specification "
        "is the ONLY thing they get. Therefore, derived PURELY from what the tests "
        "exercise:\n"
        "- Describe the required behavior -- expected inputs, outputs, edge cases, error "
        "conditions -- "
        "for exactly what the tests check.\n"
        "- NAME the public API surface the tests import and call: module paths, class "
        "names, "
        "function/method names and signatures (the engineer must create these so the "
        "hidden tests "
        "can import and call them).\n"
        "- Do NOT reference the tests themselves: no test file/function names, no "
        "assertion snippets, "
        "no 'the test expects...'. Describe the BEHAVIOR the tests imply, not the "
        "tests.\n"
        "- Do NOT describe the implementation -- state WHAT is required, not HOW.\n"
        "- Do NOT add behavior the tests don't cover, even if commit messages/issues "
        "mention it.\n\n"
        f"{context}\n"
        "Write the self-contained, test-grounded problem statement now:"
    )
    result = _worker_llm_call(vllm_url, prompt, max_tokens=None)
    if result == CONTEXT_OVERFLOW or not result:
        # Tests are the source of truth -- on overflow, DROP the narrative (commit
        # messages + issues), KEEP the test diff, and retry. (Old behavior dropped
        # the tests first, which is backwards for a test-driven spec.)
        if overall_test_patch and overall_test_patch.strip():
            test_only_context = ("=== TEST CHANGES (source of truth -- describe the "
                                 "behavior they require, never reference them) ===\n"
                                 + overall_test_patch + "\n")
            result = _worker_llm_call(
                vllm_url, prompt.replace(context, test_only_context), max_tokens=None)
        if result == CONTEXT_OVERFLOW or not result:
            return [], issue_used
    return [result.strip()], issue_used
```

D.7 Environments, the repair loop and the two-container verifier

Each repository receives one image built from source, with the dependencies of every commit in the chain installed offline from a frozen closure of ≈3,000 wheels; history is scrubbed to the base and gold is never present. A task is pinned to an image digest, and it is the digest, not a rebuild, that is certified. Acceptance is a model-free *gold walk*: replaying the 20 gold patches with the hidden tests must pass at every state. 225 of 397 environments passed at once; GPT-5.6-sol (xhigh, at most 80 turns), allowed to edit only the Dockerfile and accepted only by the model-free gold walk, repaired 167 more, and the 5 unrepairable windows were dropped.

Auditing the repaired images exposed a failure unrelated to the agents under test: pytest hooks that rewrote a passing base outcome to `failed` to fake the F2P precondition, hidden node IDs baked into image layers, dependencies branching on hidden-test names, and `git apply` wrappers that special-cased the gold path. The acceptance instrument had run inside the artifact it certified. All 32 flagged environments were dropped (the release has 200 baseline and 94 repaired environments, pre-repair Dockerfiles included) and the verifier was rebuilt so this cannot recur: agent and verifier run in separate containers and only `/testbed/.git` crosses between them; the verifier starts from a fresh copy of the digest with hidden tests in a root-only layer, runs candidate tests as an unprivileged

user in disposable worktrees, and captures outcomes through a registered in-memory plugin rather than a report file.

Sealing. The source images carry the repository's full history, so the gold for every later segment is reachable from `HEAD`, and installed package metadata names the future version. Each image is sealed in an offline rebuild of its repository layer: the object store is re-created from scratch with only the history reachable from the base (not pruned, since unreachable objects linger after `git gc`), the checkout is the base with a single branch ref and no other refs, packed-refs, reflogs, commit-graph files or hooks; installed Python metadata (dist-info, SCM version files, entry points) is reset to the base version; build products, caches and stray copies of the project from the future head, and any credentials, are removed. Content that base tests need (tracked files, submodules, ignored runtime assets) is kept and declared in a root-owned fingerprint that the grader trusts, so nested `.git` markers planted by an agent gain nothing. Verification runs at four layers: a scripted check in a fresh container (`HEAD` equals the base, no refs beyond the branch, `git fsck` finds nothing unreachable, the worktree is byte-identical to the base tree, imports resolve into `/testbed`, metadata at the base version, no credential files); a per-image checklist worked through by the sealing agents for what a script can miss (the former head unresolvable, `git status` shows only allowed entries, `/tmp` empty, layer size plausible, image configuration unchanged, fingerprint consistent with the base tree); the gold walk on the sealed image at every state; and the verifier itself, which at grading time rebuilds the test-time `.git` from the chain start, flags unreachable commits, refuses undeclared nested repositories, and withstands the adversarial probes of Appendix D.11, including attempts to recover the gold from `.git`.

D.8 Protocol details

The agent receives the base repository in a persistent workspace with specification 1, works through shell commands inside a network-disabled container started from the pinned digest, and is told not to modify, create or delete test files because the verifier uses frozen grader-owned tests and discards test edits. On submission only `/testbed/.git` is copied to a fresh, digest-identical verifier container, which restores grader-owned paths, applies hidden test patches $1..k$ and runs the exact contract of state k. On a pass the same conversation, never truncated, receives an acceptance message and specification $k{+}1$. The first failed segment ends the attempt; reward is 1 iff all 20 states pass; the horizon is not disclosed. Infrastructure never consumes an attempt. The diagnostics of Sections 4–5 run on the same substrate and differ only after a failure: they continue silently past a failed state, return a failure verdict with one repair opportunity, or restart with fresh context on the same state.

D.9 Grader rules

The verifier anchors at the base SHA and requires exactly 20 direct-parent commits, one per segment with the prescribed subject; merges, extra or duplicate commits, replace refs, grafts and changed git controls are rejected. At state k it applies frozen test patch k and checks that the F2P nodes do *not* already pass, then applies checkpoint diff k alone and requires every F2P_k and P2P_k node to pass literally, together with every earlier F2P node that the human gold still satisfies at k. Behavior the human history itself later superseded (24.5% of earlier F2P nodes by state 20; 255 of 280 tasks have some) is recorded and not demanded. contract applicable at k already passes before it. The grader fails closed on missing or duplicate nodes, non-zero exits and non-literal outcomes; protects test paths, `conftest`, `sitecustomize`, `.pth` and pytest configuration; and accepts security-sensitive changes only when byte-identical to gold.

D.10 Filtering rounds

Once frozen, nothing is edited: a task that fails any check is dropped and archived, and a pass on rerun proves nondeterminism, never acceptability. Fifteen rounds took 439 windows to 280: 42 for contested specifications, provider-filtered or malformed chains, and re-windowing (R1–R3); 32 for the environment manipulation above (R8); 27 for specification hygiene, gold patches touching test-classified paths, or non-reproducible repaired environments (R5, R6); 17 for gold-walk, packaging, apply and window-composition failures (R4, R7); 20 for failing the first two-container gate (R12); 4 for nondeterminism on an identical digest (R9–R11); 2 for verifier time above 1,800 s (R13); 1 for a

frozen node failing under the final grader (R14); and, last, 14 chains in six repositories identified post hoc from the evaluation runs (R15): in each, a model failed the same hidden test when it started from its own implementations of the earlier segments but passed it when it started from the human ones, so the test presupposed the human implementation of an earlier segment rather than its specification. The benchmark is 280 chains, 140 repositories and 4,404 distinct segments.

D.11 Certification and threat model

The 280 tasks pass a digest-bound gold walk at all 20 states, the final grader gate and an independent no-retry repeat; the Oracle scores 1 through the same two-container path an agent uses. Twenty adversarial canaries score 0: payloads committed before the chain, extra, duplicate or merge commits, replace refs, grafts, hooks, test renames or edits, symlinks, gitlinks, grader-aware source and grader tokens, among others. Five legitimate variants score 1: gold, an untracked conftest, uncommitted source edits, tampering with an external pytest, and an already-satisfied empty checkpoint. The residual limit is stated: no executable-test benchmark can stop an implementation from branching on functional inputs. After the fact, 28,212 agent trajectories covering 483,417 graded steps were scanned with 14 detector families and a judged sample; no grader defeat was confirmed.

D.12 Release

BOBA-Bench contains 280 tasks from 140 repositories, 4,404 unique segments and 5,600 graded states; task images are mirrored to an immutable archive registry identified by SHA-256. Temporal exposure and memorization are measured, not assumed absent (Appendix E).

E Prior Exposure and Memorization

Direct access and prior exposure are different. The evaluation image contains the base checkout and no reachable future repository history. The seal removes future git objects, refs, reflogs, hooks, caches, and credentials, and resets source-control version metadata to the base. Scripted checks and the reference-patch walk test the sealed image (Appendix D.11). These controls prevent direct retrieval through the evaluation environment; they do not erase code learned before evaluation.

Source dates. Among the dated model cutoffs we compared, only one of the benchmark's 4,404 unique requests postdates the earliest, and none postdates the others. that the benchmark is unseen.

Patch-continuation probe. The model receives the first 40% of a reference-patch hunk and attempts its remainder. The near-verbatim endpoint uses a 0.9 similarity threshold. Fable 5 reproduces 15.1% of benchmark hunks near-verbatim (interval 8.9–22.2%) and 14.5% of same-repository hunks from before its cutoff (Table 18). Its matched contrast is $+0.6$ points (interval $[-8.5, +9.7]$). Nearly all benchmark requests also predate the model cutoffs, so the probe cannot rule out exposure to either set. Table 18 gives all four probed models; the other three reproduce at most 1.2% of sampled hunks.

We report the probe as limited evidence about reproduction, not proof of no prior exposure.

F Evaluation Details

F.1 Primary run and harness comparison

The primary harness is mini-swe-agent (Yang et al., 2024). Each chain attempt permits unlimited model calls, two hours of cumulative tool execution, and twelve hours of wall time. Fable 5 ran at the provider's default reasoning setting, GPT-6 Astra and GPT-5.6 at `xhigh` reasoning, GLM-5.2 at temperature 1.0, and Kimi K3 was served with SGLang. These settings do not equalize compute across models.

Infrastructure failures trigger a fresh attempt from state one, and provider content filters are retried. Nevertheless, repeated infrastructure failures can exhaust a scheduled slot. Appendix F.2 separates those no-verdict outcomes from hidden-contract failures.

Table 37 compares scaffolds on a stratified subset of 58 chains from 52 repositories: a 60-chain draw from the release minus the two chains of an excluded repository. No paired difference excludes zero. The intervals rule out scaffold effects above about 21 points, not smaller ones. Most chains get the same verdict under all three scaffolds of a model: 44 of 58 for GPT-5.6 and 48 of 58 for Kimi K3. GLM-5.2's OpenHands SDK run ended 29 of its 60 attempts at the wall limit and is not comparable.

F.2 Chain completion per model

Table 2 gives the per-model figures behind Figure 1 and Section 4.1. The middle columns give the share of first attempts that reach request k (reached@k) and the number of requests a first attempt completes, a complete chain counting 20. The last column averages, across a model's first attempts, the share of submissions that failed their hidden contract. First attempts without a verdict count as failures. Wilson intervals for pass@1 and the pairwise tests are in Appendix H.

F.3 Resume arms: per-model results

Table 19 gives the per-model estimates behind Section 4.2. Starting points are every request a model submitted as complete that failed its hidden contract ($k \leq 19$); the control is the benchmark's own continuation after a certified pass of the same request, standardized to the arm's distribution of k. Own-work failure at $k{+}1$ counts failures of $k{+}1$'s own tests and excludes failures inherited from the open defect. Intervals are Wilson 95% for rates and repository-clustered bootstrap 95% for contrasts (resampling arm and control together); the repaired and unrepaired subsets of arm E are outcome-selected. GPT-6 Astra and GLM-5.2 have no arm F.

F.4 Own-work failure by number of open defects

Table 20 pools every graded request from $k{+}1$ to 20 of the silent continuations of Section 4.2 and groups them by the number of defects still open beneath the request when it was issued (the original miss plus any later own-work failures). The odds ratio is the slope of a logistic fit of own-work failure on the count (capped at five), with a 95% repository-bootstrap interval. Requests with many open defects sit late in the chain and come from continuations that were already failing, so the progression is descriptive, not a controlled dose.

Estimated cost. The cost estimate was computed before the last filtering round of Appendix D.10 (the per-check quantities below do not depend on it). Estimated verifier compute is check, with a median runtime of 16–17 seconds at four virtual CPUs. A cost model with a check after every request and one repair after each failure gives 7–14% expected overhead for Fable 5, GPT-5.6, Kimi K3 and GLM-5.2; GPT-6 Astra is not included. The model uses list prices retrieved on 9 September 2026, without prompt caching. This is a modeled overhead, not a measured human-review cost, and it excludes test construction and maintenance.

F.5 Noisy supervisor: intervals

Figure 6 plots the noisy-supervisor results in Table 33 with repository-clustered 95% bootstrap bands.

F.6 Root causes of silent misses

Every silent miss of Section 4.1 is a request the agent submitted as complete that failed its hidden contract. To ask why, we labeled every first-attempt silent miss of Claude Fable 5, GPT-5.6, GLM-5.2 and Kimi K3 on the 140 repositories, 575 sites in all (89, 156, 188 and 142; GPT-6 Astra's run finished after the census). A judge model (GPT-5.6-sol at its highest reasoning setting) reads, for each site, the specification as delivered, the agent's diff and final message, the failing hidden tests with their output, the hidden test patch, the gold patch and the compacted trajectory, and assigns one of 17 failure tags adapted from the DeepSWE taxonomy. A second judge (Claude Fable 5) labeled a stratified sample of 300 sites (275 on the 140 repositories): the two choose the same tag on 55% of them, κ 0.46 [0.39, 0.52], and the same coarse class on 58%; where they differ they describe the

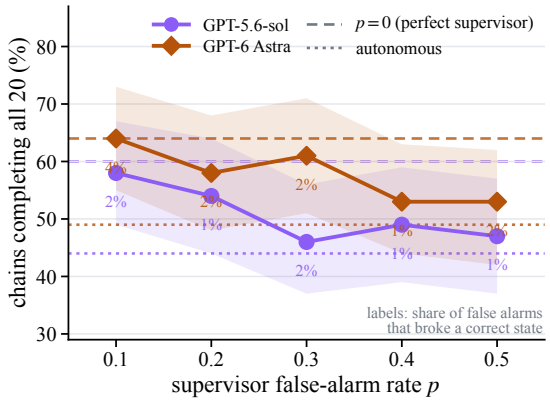


Figure 6: **A noisy supervisor, with intervals.** Noisy-supervisor results from Table 33: completion at 20 against the supervisor's false-alarm rate p on 100 chains (GPT-5.6-sol, GPT-6 Astra; $N{=}100$ per level), with repository-clustered 95% bootstrap bands. Dashed: perfect supervisor ($p{=}0$); dotted: autonomous. Labels: share of false alarms that broke a correct state.

same mechanism and disagree on the tag. The labels are therefore indicative, not exact. Table 21 gives the distribution; the four paragraphs below describe the agent-side causes.

F.6.1 Regressions

Regression is the modal class for every model (26–29% of its misses): the agent implements the request and breaks behavior the request did not ask it to change. In 140 of the 158 regression sites a pre-existing test (a P2P node) fails; in the remaining 18 the new hidden test exercises pre-existing behavior that the edit broke. The share is flat over the chain (27%, 33% and 21% for requests 1–7, 8–14 and 15–20).

F.6.2 Missed requirements

One miss in five leaves part of the specification unimplemented: the tests of the new behavior fail and nothing else does (93% of these sites fail only F2P nodes). The class separates the models more than any other: 8% of Claude Fable 5's misses against 26% of GLM-5.2's. It is also the only class that grows along the chain (17%, 17% and 27% by band), so late in a chain the weakest models increasingly omit what was asked.

GLM-5.2 explicitly anticipated that its change would break specification 8 and treated that as acceptable; the subsequent submission failed. In the traced example (Appendix I.3), it changed the decorators while acknowledging the loss of request 8's behavior, and two tests from request 8 then failed.

F.6.3 Wrong logic and integration errors

A further 18% of misses implement the request but wrongly: the new code computes the wrong result (wrong logic, 10%) or is correct but never reached, wired into a literal but not into the constructor path the tests exercise, or into a sibling of the function the request names (integration error, 8%). Both are F2P-only failures in nearly every case. Integration errors are rarest for GPT-5.6 (3%) and most common for GLM-5.2 (12%).

F.6.4 Unverified submission

The self-verification census (51,188 submitted requests of Claude Fable 5, GPT-5.6 and GLM-5.2 on the 280 chains) asks what the agent did before it submitted. A green local test run does not lower the chance of a miss: submissions preceded by a passing run fail the hidden contract 4.8%, 6.3% and 3.8% of the time against 4.7%, 6.1% and 3.9% when no test runner was invoked at all, and 48%,

77% and 61% of misses follow a clean local run. test what they wrote, not what was asked; a passing run of their own tests is what makes them believe a request complete.

F.7 Additional leaderboard details

Complete-chain rates. Table 22 gives pass@1 and pass@8 on the 280 chains with repository-clustered 95% bootstrap intervals.

First attempts without a verdict. A first attempt counts as a failure in pass@1 whether or not the hidden grader ever returned a verdict on it. Two kinds of attempt have none (Table 23). With *no valid record*, the slot produced no graded attempt before the 12 h deadline, because harness errors kept being retried from state 1. An attempt *ended by a budget* hit the 12 h wall, the 2 h tool cap, or the provider's context limit; every wall-limit case followed 17 to 257 infrastructure retries. Counting all of these attempts as failures is conservative. In the mean states passed, such an attempt contributes the states it had passed, or none when no record exists. Of the fourteen GLM-5.2 wall terminations, nine had passed no state and the other five had passed 4 to 17. In the silent-miss rate, such an attempt contributes its accepted submissions and no failure, since none of its requests failed a contract. Excluding these attempts instead raises the Fable 5 mean from 16.3 to 16.6 states and changes no other model's mean by more than 0.05.

Survival over the chain. Table 24 gives reached@k, the share of first attempts that had passed state k, for $k \in \{5, 10, 15, 20\}$. The sum of the curve over $k = 1..20$ equals the mean number of requests completed in the same table. The table also gives the per-request miss rate by quarter of the chain (contract failures over submissions in the quarter); the pooled rate is the silent-miss rate of Table 2.

Retries and the constant-rate fit. A chain gets up to eight attempts and is not re-run after its first pass, except for GLM-5.2, which ran all eight on every chain. The second attempt passes 8–19% of the chains whose first attempt failed (13 of 89 for Fable 5, 11 of 134 for GPT-6 Astra, 27 of 144 for Kimi K3, 26 of 157 for GPT-5.6 and 17 of 202 for GLM-5.2; denominators are the chains with a scored second attempt), and seven further attempts rescue 19–45% of first-attempt failures (39 of 96 for Fable 5, 25 of 134 for GPT-6 Astra, 65 of 145 for Kimi K3, 51 of 157 for GPT-5.6 and 54 of 202 for GLM-5.2). GLM-5.2 never passes 148 chains, always passes 27 and sometimes passes 105; if every chain had its pooled 25.6% chance per attempt, about 26 chains would fail all eight attempts and none would pass all eight. With h the unrounded silent-miss rate of Table 2, $(1-h)^{20}$ predicts completion of 67.9, 53.3, 48.2, 45.9 and 30.7% against the observed 65.7, 52.1, 48.2, 43.9 and 27.9%, so it overestimates four models by 1.2–2.8 points and matches Kimi K3 to within 0.1 point; per model, the largest gap between $(1-h)^k$ and reached@k is 2.2–5.4 points.

Shared outcomes. On the first attempt, 70, 38, 34, 43, 44 and 51 chains are completed by 0, 1, 2, 3, 4 and 5 models; if models failed independently at their own rates, about 9.6 chains would be completed by none and 5.7 by all five. When two models both fail a chain, they fail at the same request in 53.5% of cases (537 of 1,004 pairs), against 5.2% if failure positions were independent. Any two models agree on pass or fail for 58–77% of chains (chance: 43–53%), and the 42 chains that defeat every model in eight attempts come from 33 repositories, with a median of 4–8 requests completed.

Clustered intervals. Chains from one repository share code, tests and history, so every interval in Table 2 resamples the 140 repositories with replacement (10,000 draws), recomputes the statistic as a ratio of sums over the resampled chains (solved chains over chains, states over chains, failed submissions over submissions) and reports the 2.5th and 97.5th percentiles. The 140 repositories contribute one to five chains each (34, 81, 18, 5 and 2 repositories respectively).

G Reproducibility and Evidence Scope

What the paper documents. Appendix D gives the construction pipeline, the specification prompt, grader rules, and every filtering round; Appendix F gives models, settings, and budgets; Table 25 gives the population of each study.

Replication. A replication needs the task images by digest, the grader, the specifications, and the per-attempt verdicts, together with the model and harness versions of Appendix F.

Statistical scope. Baseline intervals use 10,000 repository bootstrap draws. Task-statistics intervals use 2,000 draws. Pairwise completion intervals use 20,000 draws. These counts belong to different analyses, not inconsistent repeats of one procedure. The prefix, review, and repair tables use the intervals and tests of each study. We infer no paired uncertainty from marginal rates.

H How well do coding benchmarks separate models?

This appendix documents the computation behind the separation claims of Section 2: that no two of the top five models on SWE-Milestone or SlopCodeBench can be told apart from the counts those papers report, that SWE-Chain reports no dispersion at all, and that on BOBA-BENCH eight of the ten pairs among five models are separated, the leader from every other model at $p < 10^{-5}$. The script (`separation_analysis.py`) and its output accompany the paper.

Data. For each benchmark we take the per-model figures from the paper's main results table and the unit the figure is a fraction of (Table 26). Where the table reports a percentage rather than a count we recover the count as $\mathrm{round}(\text{rate} \times n)$; for SlopCodeBench the denominator is fixed at 196 by construction, and for SWE-Milestone the rates are per-evolution-range averages, so the implied count is approximate to within one milestone. SWE-Milestone's primary metric, Score, is a mean of recall×precision and not a fraction of trials, so we use its Resolve Rate (a milestone is fully resolved or not). SWE-Chain's resolving rate is a mean over 12 chains of test-level micro-averages from a single run, with no per-chain values or dispersion published; it cannot be tested, and we show binomial bounds for orientation only. For BOBA-BENCH we use attempt-1 outcomes on the 280-chain set of Section 3; a chain whose attempt-1 record is missing (five for Fable 5, two for Kimi K3) counts as a failure, as in the published rates.

Tests. Intervals are Wilson 95% score intervals. For the other benchmarks the papers publish aggregates, not per-task outcomes, so two models can only be compared with an unpaired test: we use the two-sided Fisher exact test on the two counts at $\alpha = 0.05$. For BOBA-BENCH every model ran the same 280 chains, so we use the exact two-sided McNemar test on the discordant chains—the test Liu et al. (2026b) apply to adjacent SWE-bench Verified entries—and, because the 280 chains come from 140 repositories (one to five chains each), a repository-clustered paired bootstrap of the difference in pass@1 (repositories resampled with replacement, all their chains entering, 20,000 resamples, seed 20260921). We also report the unpaired Fisher test on our own counts so that the comparison with the other benchmarks uses the same test. No multiplicity correction is applied on either side; a correction would only reduce the number of separated pairs, and does not change which claims below hold.

Other benchmarks. Table 27 gives, for each benchmark, how many of all model pairs, of adjacent pairs in the ranking, and of pairs among the top five are separated. On none of them is any adjacent pair or any pair among the top five separated. Separation exists only between a leader and clearly weaker models: on SWE-Milestone Gemini 3 Pro at 13.4% is separable only from MiniMax M2.5 at 1.3%; on SlopCodeBench GPT 5.5 at 14.8% is first separable from Sonnet 4.6 at 7.1% (strict) or Kimi K2.6 at 18.9% from 28.1% (isolated). On SWE-bench Verified, Liu et al. (2026b) find that exact McNemar tests separate none of the 29 adjacent pairs among the top thirty entries, whose scores span 8.8 points.

BOBA-BENCH. Table 28 gives the five models' pass@1 with Wilson intervals and Table 29 every pairwise comparison. Eight of the ten pairs are separated by the McNemar test and by the clustered bootstrap (the same eight), seven by the unpaired Fisher test; two of the four adjacent pairs are separated. Fable 5 is separated from every other model ($p \le 5.9 \times 10^{-6}$; $p \le 1.1 \times 10^{-6}$ if chains with a missing attempt-1 record are excluded instead of counted as failures). The two pairs that are not separated are GPT-6 Astra vs. Kimi K3 (+3.9 pp, $p = 0.25$) and Kimi K3 vs. GPT-5.6 (+4.3 pp, $p = 0.18$); the ranking therefore has three tiers, Fable 5, then GPT-6 Astra, Kimi K3 and GPT-5.6, then GLM-5.2. The number of models differs from the other leaderboards (5 against 9–15), so the

comparable statements are about adjacent pairs, pairs among the top five, and the leader against the field, not the raw count of separated pairs.

What the comparison does and does not establish. Three caveats bound the claim. First, the other benchmarks are tested unpaired because per-task outcomes are not published; with those outcomes a paired test could separate more pairs than we find, which is why Section 2 says "from their reported counts". Second, treating tasks, milestones or checkpoints as independent trials ignores their clustering within repositories or problems (SWE-Milestone's 98 milestones come from seven repositories, SlopCodeBench's 196 checkpoints from 36 problems); the true intervals are therefore wider than those in Table 27, which makes the finding of no separation conservative in that respect. Third, the leaderboards evaluate different model sets under different scaffolds, so the comparison is about the resolving power of each benchmark on the models its authors ran, not about any particular model.

I Trial Records: Failure Anatomy, Self-Checks, and Traced Misses

This appendix gives the long form of three results of Section 4.2 ("How agents miss"). It reads the per-request grader records and the agent transcripts of the first attempts on the 280 chains, and each analysis uses the first attempts with a finished per-request record. Unless stated otherwise, intervals are repository-clustered bootstrap 95% intervals.

I.1 Failure anatomy

Method. The contract of state k holds the request's fail-to-pass tests ($F2P_k$) and its pass-to-pass tests ($P2P_k$; Appendix D.2). The per-request grader records list the collected and the failing tests but give only the sizes of $F2P_k$ and $P2P_k$. We therefore rebuild the split from the test captures of the reference run, in which the human patches pass every contract. A contract test is in $F2P_k$ if it does not pass, or its file does not import, at the base of state k with request k's test changes applied, before the reference patch of request k; every other contract test is in $P2P_k$. The rebuilt contract equals the grader's collected list in 84,836 of 84,967 passing per-state records (the rest differ by one redacted file name in two repositories), and every failing test of every failed state lies inside it. The rebuilt $F2P_k$ matches the grader's F2P count in 5,174 of the 5,458 reached states, and in 61 more once the tests hidden by a collection error at the base are added.

A failing test counts only if the grader's run gives evidence for it: the test ran and did not pass, its file failed to import, or it is missing from a run that raised no collection error. A test that never ran because another file stopped the run does not count. A chain-ending failure is *own only* if only $F2P_k$ tests fail, and a *regression* if at least one $P2P_k$ test fails. Every class comes from grader output; no model judges it. Every classified failure followed the agent's own submission, so each one is a silent miss. Appendix F.6 labels the first-attempt misses of four models with a judge model instead and reports a lower regression share (27.5% of 575 misses) by a different method: the judge assigns one cause per miss and can place a failing pre-existing test on the benchmark side, while this classification counts every such failure as a regression.

Unclear cases. A further 59 failures have no class, because no contract test ran or because a failing test sits at a state whose split could not be rebuilt (223 of the 5,458 reached states; an independent parser leaves 188). Counting all or none of them as regressions keeps the pooled share within 30.2–38.6%. Where the reference base run stopped on a collection error, a test that is new at request k can be labeled $P2P_k$. At most seven regressions rest only on such tests, and dropping them gives 32.2% [26.1, 38.4]. An independent re-implementation of the split and the classes gives 33.3% (215 of 646) [27.4, 39.5].

Where and how far back. The regression share is 23.3% [14.2, 33.0] of failures at requests 1–5, 40.2% at requests 6–10, 43.4% at requests 11–15, and 24.6% at requests 16–20. The rise from requests 1–5 to requests 6–15 is 17.7 points [4.8, 30.6], but the share falls again at the end of the chain, so we do not claim a monotone trend. Of the 212 regressions, 107 (50.5% [39.7, 61.0]) break a test that an earlier request of the chain added or changed; about three in five of these tests were

new at that request, and the others were existing tests that the request changed. The nearest such request lies a median of two requests back (interquartile range 1–5), and 31.8% [19.8, 45.3] of these regressions break a request five or more back. In 70.8% [59.7, 81.1] of all regressions, the same attempt had passed a broken test at an earlier state. In 24.6% [18.9, 30.5] of classified failures, every $F2P_k$ test passes and only earlier behavior breaks, so a grader of the new request's tests alone would accept the submission. Over all graded attempts, not only first attempts, the regression share is 34.0% [27.8, 40.1]. The contract checks only tested behavior. Breakage of untested behavior is not measured, so the true regression share is likely higher.

I.2 Self-checks before submission

Sample. From first attempts with a finished record on the 280 chains, we drew, per model, 60 silent misses and 60 accepted submissions uniformly at random with a fixed seed: 600 submitted and graded requests from 525 attempts in 136 repositories. A miss is the failed request that ended its attempt, so the 300 misses come from 300 different attempts.

Rules. Fixed regular-expression rules read only the sampled request's commands and tool outputs; no model labels the reported rates. The agent *ran tests* if it called a test runner (pytest, unittest, tox, nox, or a `make` test target) or ran a Python assertion script that writes no repository file. It *ran the repository suite* if a runner call had no targets, or only repository paths that the agent had not written in that request. *Wrote test* counts writes of test files, scratch files included. The *last run* is the outcome of the last test command. It failed if a runner summary reports failures or errors, a test is listed as failed, or a script raises a traceback or exits with an error; it passed if a summary reports passes and no failures, or a script exits cleanly; otherwise it is unknown. The agent *claimed success* if one of its last three non-empty messages states success (for example, that all tests pass or that the work is verified or complete) and the final message has no hedge. *Green, then claimed* requires a test run, a passing last run, and a success claim.

Rule check. We checked the rules against labels that a language model assigned from the full transcripts, on requests not used to write the rules. On 20 such requests, the action rules agreed on 100 of 100 labels and the last-run outcome on 18 of 20. After one revision, the success-claim rule agreed on 18 of 20 further requests. An independent re-implementation of the rules gives 100% for running tests, 82.0% for the repository suite, and 56.3% for green, then claimed, against 99.7%, 85.0%, and 61.3% in Table 31.

Results. Because the sample is stratified on the outcome, we compare misses with accepted submissions through Mantel–Haenszel odds ratios across models. Neither running the repository suite (1.44 [0.90, 2.29]) nor a passing last run (1.33 [0.91, 1.98]) is associated with fewer silent misses. In half of the green-then-claimed misses (93 of 184), the passing last run was the repository suite; in the rest it was the agent's own script or a targeted run. In at least 43 of the 300 misses (14.3%, Wilson [10.8, 18.8]), the agent's own test output named a test that the grader later failed, and the agent's last mention of that test still reported a failure. This count is a lower bound: agents often print only summary lines, and the records keep at most 50 failing test identifiers per request. Appendix F.6.4 counts every submitted request of three models instead of a sample and finds that 48%, 77% and 61% of the misses of Fable 5, GPT-5.6 and GLM-5.2 follow a clean local run; that census applies its own test for a clean run and requires no success claim, so its shares are not directly comparable with Table 31.

I.3 Two traced misses

We chose these two first-attempt silent misses for clarity, not at random. Both meet five conditions: the specification states the behavior; the agent tested its change and declared success; the failing hidden test shows what was missed; that test belongs to an earlier request; and another model or attempt passed the same request. Quotes are verbatim, square brackets mark our edits, and code is abridged. One quote in Case 1 comes from the model's reasoning trace, which is part of the attempt record.

Case 1: `python-poetry/cleo`, GLM-5.2, request 9 (Table 32). Request 8 adds three decorators to `Application`. Each registers a command. Used alone, `@app.argument` and `@app.option` register the command under the decorated function's name. The agent's own script for request 8 checks `app2.has('arg_function')`, and the request passes. Request 9 asks that `@command`, `@argument` and `@option` stack on one function, with `@command` applied last. It states: "The `argument` and `option` decorators contribute definition entries; the `command` decorator finalizes registration." It says nothing about a decorator used alone. The agent rewrites `argument` and `option` so that they only store a definition on the function for `@command` to collect. Its reasoning trace notes the conflict: "For standalone use of `argument` and `option` (from the previous [request]), they would NOT create commands anymore. But that's OK because the new spec redefines their behavior." It searches the visible tests for decorator tests, finds none, and does not re-run its request-8 script. Each of its checks uses the stacked form; the last prints `ALL SPEC REQUIREMENTS VERIFIED!`. After 28 model calls it submits with the claim "No new test failures were introduced — only the 2 pre-existing progress helper failures remain." The one new test of request 9 passes. Two of the 197 carried-over tests fail: `test_argument_decorator` and `test_option_decorator`, both from request 8, at `self.assertTrue(application.has('decorated_foo_code'))` with `AssertionError: False is not true`. A correct change keeps the request-8 behavior for a decorator used alone and adds the stacked behavior. GPT-6 Astra, Kimi K3 and GPT-5.6 pass request 9 on their first attempts, and GLM-5.2's second attempt fails the same two tests. Across all graded submissions, 4 of 6 pass request 9.

Case 2: `HewlettPackard/oneview-python`, GPT-5.6, request 13. Request 12 adds `delete_bulk(resource, timeout=-1)` to `FcNetworks`, `EthernetNetworks` and `FcoeNetworks`. Request 13 makes bulk delete independent of a loaded network for the Ethernet and FC clients. It states: "Do not extend the identifier-free requirement to `FcoeNetworks.delete_bulk`; FCoE receives only the specified whitespace cleanup in this task," and "Preserve all existing public APIs, signatures, defaults, and behavior not explicitly changed above." The agent deletes the five lines of `FcoeNetworks.delete_bulk`; this deletion is its whole submission. Its check asserts `not hasattr(FcoeNetworks, 'delete_bulk')` and prints `collection-level bulk delete contract checks passed`; the visible network tests report `32 passed`. After five model calls it explains: "I removed the out-of-scope FCoE bulk-delete method as required." Both new tests of request 13 pass. One of the 1,296 carried-over tests fails: the request-12 test `FcoeNetworksTest::test_delete_bulk`, with `AttributeError: 'FcoeNetworks' object has no attribute 'delete_bulk'`. A correct change leaves the FCoE method in place. The four other models pass request 13 on their first attempts, and GPT-5.6 passes it on its second. Across all graded submissions, 10 of 11 pass request 13.

J Supporting Details for Related Work

This appendix retains quantitative and construction details omitted from Section 2. These details describe the evaluation setting and its controls; they do not establish that tasks across benchmarks have equal difficulty or that a larger benchmark necessarily distinguishes every model pair.

Counting requests consistently. We count one separately delivered specification document as one request, without splitting its internal requirements into additional tasks. SWE-Milestone contains 98 graded specifications and three additional, ungraded specifications that agents still implement (Deng et al., 2026). SWE-Chain contains 155 version-transition specifications, each bundling upgrade requirements (Lam et al., 2026). SlopCodeBench contains 196 checkpoint specifications across 36 problems (Orlanski et al., 2026). Including the ungraded SWE-Milestone requests gives 452 specifications in total. These counts were checked against SWE-Milestone dataset revision `3500478e`, SWE-Chain dataset revision `7080d652`, and the SlopCodeBench problems release `v1.0` (`4d38d300`). The corresponding source snapshots and counting scripts accompany our local research artifacts.

BOBA-Bench contains 280 chains of 20 requests across 140 repositories, yielding 5,600 request occurrences but 4,404 distinct specifications. The final window of a retained run overlaps the pre-

ceding window when needed to include its tail, producing 1,196 repeated occurrences at different chain positions. The distinct-request ratio is therefore $4{,}404/452 = 9.74$, not the occurrence ratio $5{,}600/452 = 12.39$. Repeated requests have different prefixes but are not independent observations for repository-level inference. These are counts of specification documents, not comparable amounts of implementation work.

Task-quality evidence. OpenAI's SWE-bench Verified audit establishes a known-defect floor above 16% of the full benchmark, based on an audit that targeted frequently failed tasks rather than a random sample (OpenAI, 2026d). Its SWE-bench Pro analysis reports defects in roughly 30% of the public split, with the exact proportion depending on the audit procedure (OpenAI, 2026c). These figures concern different auditing procedures and should not be interpreted as directly comparable prevalence estimates. Other studies identify solution leakage, weak tests, and opportunities for exploitation (Aleithan et al., 2024; Yu et al., 2025; Zhong et al., 2025).

SWE-Milestone generates requirements from reference patches, aligns them with tests, and uses expert review to check solvability and implementation leakage (Deng et al., 2026). BOBA specification generation receives test changes and development descriptions, but not the source patch. This removes direct access to the reference implementation at generation time, rather than proving that every resulting specification is complete or free of hints. Appendix D.4 describes generation and reference-code validation.

The isolated-success figure formerly quoted in Related Work concerns sampled requests, not every request in every chain. For Claude Fable 5, 1,089 of 1,120 scheduled attempts on one sampled request per repository passed, or 97.23%, counting the 28 ungraded attempts as failures. Among graded attempts the fraction was 1,089/1,092. These denominators must not be confused with distinct-request coverage or the probability of at least one success across repeated attempts. The count comes from the filtered per-target isolated-attempt summary, not from the earlier candidate-certification campaign.

Model comparison and uncertainty. The manuscript's earlier Related Work combined illustrative binomial interval widths with claims about ranking models. Counts of repositories, chains, and checkpoints are not interchangeable statistical units. Within-repository dependence must be accounted for, and an aggregate rate cannot in general be multiplied by a nominal task count to reconstruct valid binomial outcomes. In particular, a macro-averaged milestone rate is not a binomial proportion over all milestones. The absence of published uncertainty does not demonstrate that models are indistinguishable. We therefore do not retain the earlier numerical resolution argument or the blanket non-separation claim in Section 2. The predecessor calculations in Appendix H require revision before they can support comparative significance claims. BOBA comparisons should use the observed paired outcomes and repository-clustered uncertainty, without treating a nonsignificant difference as evidence of equivalence.

K TRUST, BUT VERIFY: PROTOCOLS AND ESTIMATES

This appendix gives, for each arm of Section 5, what the agent was told, which runs were used, and the per-model estimates with intervals. Intervals are Wilson 95% for rates, repository-clustered bootstrap 95% for contrasts (2,000 replicates), unless stated.

Table 33 summarizes the section's estimates; the subsections below give counts, intervals and protocols.

K.1 ARMS AND MESSAGES

Starting points. The Reject, Name, Insist and fresh-conversation arms start from the silent misses of §4.2: every request $k \leq 19$ that a model submitted as complete and that failed its hidden contract. The failing state is restored and the same conversation is continued. Counts per model are in Table 19.

The message. In every arm the supervisor's message is the same string, sent as the observation to the agent's submit command:

> *External verification of your previous submission failed: the repository does not satisfy the previous specification. Fix the repository so that the previous specification is satisfied, then submit again with the same command. Do not modify or create test files.*

Reject (arm E) sends it once and allows one repair round before request $k{+}1$ is issued. *Name* (arm E1-names) appends the identifiers of the failing test nodes. *Insist* (arm E8) re-sends it after each failed repair, for up to eight rounds. *Fresh conversation* (arm F) restores the same broken state into a new conversation with no message, as a control for the effect of a restart. The next request is graded on its own tests (own-work failure), so a failure inherited from the open miss is not counted against it; the reference rate is the same model's own-work failure after a certified pass of request k, standardised to the arm's distribution of k (Table 19). The repaired and unrepaired subsets are selected on outcome; their rates describe, they do not estimate a causal effect.

Gate. The benchmark's first attempts are audited from the exact state at which they first failed (identity-checked restore): one repair round is allowed at that request and at every later request that fails, including the twentieth, and a chain counts as complete only if all twenty requests pass. Kimi K3's continuations that reached the 12-hour wall while still passing are counted as failures (18 of 145 audited chains), so its completion is a lower bound; every other model's audit is fully resolved on the 280 chains.

Noisy supervisor. Fresh chains are run from the first request on 100 chains of the 280-chain set (100 distinct repositories; 44 that GPT-5.6 completed on its first attempt, 30 that it failed by request 10 and 26 that it failed later), with GPT-5.6-sol and GPT-6 Astra at the benchmark's settings. After every submission the hidden grader runs: after a failure the message above is sent (a true alarm); after a pass it is sent with probability p (a false alarm), otherwise the usual acceptance. One uniform draw per (chain, request), fixed before launch, decides the false alarms at every level $p \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, so the levels are nested. Each alarm allows one repair round, after which the verdict is final: a false alarm that makes the agent break a correct submission ends the chain. The references are the same 100 chains without a supervisor (first-attempt pass) and under the Gate audit above ($p{=}0$). A first run of this arm wrote its bookkeeping—the level, the chain's alarm schedule and the kind of each alarm—into a directory that the harness mounts into the agent's sandbox; GPT-5.6 read at least one of these files in 13–19 of 100 attempts per level and GPT-6 Astra in at most one. The arm was re-run with the bookkeeping sealed outside the sandbox and a per-attempt check that nothing of it was visible. Section 5.4 reports the sealed run; the first run is reported below as a sensitivity analysis.

K.2 Reject, one round

Table 19 gives the per-model estimates on the 280-chain set: starting points, repair rate with Wilson intervals, own-work failure at $k{+}1$ on repaired, unrepaired and certified states, and the paired contrast between the fresh-conversation and silent arms ($F - D$: Table 5 $+4.1$ pp [0.7, 8.2] on 465 pairs; Kimi K3 $+0.4$ [-1.9, 2.7] on 773; GPT-5.6 -1.2 [-5.1, 2.5] on 939; GPT-6 Astra and GLM-5.2 have no arm F).

K.3 Name and Insist

Both variants were run for GPT-5.6 and Kimi K3 on the same starting points as their Reject arm and are reported on the 280 chains, recomputed from the continuation records; Table 34 gives the estimates. Contrasts are paired on starting points and clustered by repository. Insist is intention-to-treat: a continuation censored by the tool-time cap or the wall before a pass counts as unrepaired at every round, and the arms' denominators differ slightly for Kimi K3 for that reason.

K.4 Gate

Table 35 gives the audit on the 280 chains: every failed first attempt is resumed from the exact state at its failure with the rejection and one repair round, and the chain continues under the same rule until it completes or a repair round fails.

K.5 NOISY SUPERVISOR

Table 36 gives the sealed run (its bands are drawn in Figure 6): ten arms of 100 chains, 1,000 attempts, none censored, every model call at the benchmark's reasoning setting, and no harness file visible in any sandbox. Completion intervals are clustered by chain (each chain is its own repository in this sample). We also report, per model, the outcome of every phantom round—unchanged resubmit (identical tree), changed and still passing, or broken—the calls it cost, its share of the attempt's list-price spend, and the repair rate of true alarms, which is comparable in protocol but not in population to the Reject arm (here the misses arise inside fresh attempts).

Pooled over the five levels, GPT-5.6's 2,143 phantom rounds ended in an unchanged resubmit 1.4% of the time [0.6, 2.4], in a change that still passed 97.2%, and in a broken submission 1.4% [0.8, 2.1]; GPT-6 Astra's 2,295 in 0.7% [0.0, 1.8], 97.6% and 1.7% [0.9, 2.7]. A phantom round cost a median of 8 (GPT-5.6) and 6 (GPT-6 Astra) model calls and about 6 and 3 s of tool execution. True alarms were repaired in 28.7% [22.4, 34.8] and 38.1% [27.9, 48.4] of cases, against 30.0% and 45.4% in the Reject arm's population.

Sensitivity: the first run. The first run, in which the alarm schedule was readable, gives completion within 7 points of the sealed run at every level for both models (GPT-5.6: 55, 61, 47, 42, 44%; GPT-6 Astra: 63, 60, 57, 55, 57%) and the same broken rate per phantom round (1.5% for both). Where it differs is in how GPT-5.6 answered a false alarm: 11.3% [7.8, 15.2] of its 2,145 phantom rounds were unchanged resubmits, against 1.4% once sealed; restricted to the attempts whose model never read a bookkeeping file, the first run's rate is 1.2%, so the unchanged resubmits were the exposed attempts. GPT-6 Astra, which read the files in at most one attempt per level, shows no difference (0.8% against 0.7%). One first-run cell, GPT-5.6's true-alarm repair rate of 10% at $p{=}0.5$ (5 of 50), did not recur in the sealed run (25%, 14 of 57).

K.6 SCAFFOLD SENSITIVITY

Table 37 gives the scaffold re-run used in Section 5.3 (and referenced in Section 4.1): first attempts on a stratified subset under agents installed inside the task container, with the same hidden grader and budgets as the benchmark's mini-swe-agent runs.

Table 7: A BOBA-BENCH segment next to a single-request task. Median (IQR) per instance; identical parser for all rows. SWE-bench Verified from `princeton-nlp/SWE-bench_Verified` (500); SWE-bench Pro public set from `ScaleAI/SWE-bench_Pro` (731; Go 280, Python 266, JS 165, TS 20). Specification = problem statement (Verified) / problem statement + requirements + interface (Pro) / frozen segment specification (BOBA-BENCH).

	BOBA-BENCH segment n=5,600	BOBA-BENCH chain (20 seg.) n=280	SWE-bench Verified n=500	SWE-bench Pro n=731	SWE-bench Pro (Py) n=266
gold files changed	3 (2–7)	25 (16–45)	1 (1–1)	4 (2–7)	3 (2–4)
gold hunks	7 (4–15)	–	1 (1–2)	9 (5–16)	7 (4–12)
gold lines added	50 (16–146)	2,177 (1,246–4,274)	4 (2–10)	63 (33–135)	46 (30–108)
gold lines removed	14 (3–51)	775 (442–1,836)	2 (1–4)	20 (7–52)	15 (5–41)
gold lines changed	73 (24–215)	–	7 (3–13)	94 (50–207)	74 (45–142)
new functions	1 (0–3)	–	0 (0–0)	0 (0–0)	1 (0–3)
test files changed	1 (1–2)	–	1 (1–1)	1 (1–2)	1 (1–2)
test lines added	31 (12–79)	1,197 (803–1,904)	15 (7–27)	46 (17–116)	40 (14–99)
F2P tests	2 (1–7)	106 (67–187)	1 (1–2)	3 (1–7)	4 (2–8)
P2P tests	45 (22–91)	–	50 (19–111)	0 (0–30)	12 (1–51)
specification words	493 (309–780)	11,502 (9,583–13,696)	143 (85–234)	418 (320–542)	425 (318–573)
P2P non-empty	96.0%	–	97.8%	49.1%	77.8%

Table 8: Stratification bins (segment occurrences per bin; assignments in `data/segment_bins.jsonl`). Size terciles are computed over the 5,600 occurrences; position blocks are balanced by construction.

stratum	bins	n
gold lines changed (terciles)	small < 36 / medium 36–144 / large ≥ 145	1,832 / 1,896 / 1,872
source commits per segment	1 / 2–4 / ≥ 5	1,741 / 2,081 / 1,778
P2P protection	empty / non-empty	223 / 5,377
symbol/import dependency ($k \geq 2$)	none / on $k-1$ / only on $j < k-1$	1,421 / 1,533 / 2,366
position block	1–5 / 6–10 / 11–15 / 16–20	1,400 / 1,400 / 1,400 / 1,400

Table 9: **Matched Foundations per model** (140 repositories). Hidden-tests protocol; 95% repository-bootstrap CIs in brackets.

	Fable 5	GPT-6 Astra	Kimi K3	GPT-5.6	GLM-5.2
Boundaries (repositories)	311 (124)	298 (124)	332 (130)	283 (121)	286 (120)
Repositories never certified (of 140)	46	55	40	64	63
Success, agent prefix (%)	92.8 [89.0,96.2]	94.0 [90.5,97.0]	94.6 [92.1,96.9]	94.1 [91.0,96.8]	89.6 [85.6,93.2]
Success, human prefix (%)	95.1 [92.3,97.4]	93.2 [89.7,96.2]	95.1 [93.2,96.7]	93.7 [91.1,96.0]	92.2 [88.8,95.1]
Gap, agent − human (pp)	−2.3 [−5.2,+0.3]	+0.8 [−1.9,+3.3]	−0.5 [−2.7,+1.6]	+0.4 [−2.2,+2.8]	−2.6 [−5.2,−0.2]
Sign-flip p	0.11	0.59	0.67	0.76	0.04
pass@8 gap (pp)	−3.6 [−7.1,−0.5]	+0.0 [−2.7,+2.7]	+1.3 [+0.3,+2.6]	−1.0 [−4.1,+1.8]	−1.1 [−3.5,+0.7]
Gap after 4 requests (pp)	−1.4 [−4.1,+1.0]	+1.7 [−1.0,+4.6]	−1.3 [−3.6,+0.6]	−0.4 [−2.8,+1.8]	−2.4 [−4.9,−0.3]
Gap after 9 requests (pp)	+1.4 [−2.4,+5.2]	+2.0 [+0.0,+4.9]	+1.0 [−1.5,+3.9]	+3.8 [+0.1,+7.9]	−1.9 [−6.5,+2.3]
Gap after 14 requests (pp)	−3.3 [−7.9,+1.1]	−1.4 [−7.5,+4.2]	+2.8 [−0.7,+7.0]	+1.7 [−4.8,+8.7]	+0.8 [−2.5,+4.4]
Gap excl. pre-solved (pp)	−1.4 [−4.0,+1.0]	+0.2 [−2.4,+2.5]	−1.1 [−3.3,+0.8]	−0.0 [−1.7,+1.7]	−2.4 [−4.9,−0.2]
Pre-solved boundaries excluded	50	38	52	34	40
Boundaries 8/8 from both prefixes (%)	85	91	79	86	72
Boundaries with agent $\leq$ human − 4/8	8	5	4	6	7

Table 10: **Ablation: human tests of the accepted requests visible in both prefixes versus hidden** Paired on the boundaries complete under both conditions; pp with 95% repository-bootstrap CIs; gap = agent-minus-human next-request pass rate.

	Fable 5	GPT-5.6	GLM-5.2	Kimi K3
Paired boundaries (repositories)	296 (121)	280 (120)	230 (104)	263 (108)
Agent arm, visible − hidden	+5.4 [+2.1,+9.3]	+4.0 [+1.7,+6.9]	−4.1 [−7.8,−0.2]	−0.4 [−3.1,+2.4]
Human arm, visible − hidden	+3.9 [+1.5,+6.8]	+4.4 [+2.6,+6.5]	−5.1 [−8.1,−2.1]	−0.9 [−3.0,+1.1]
Gap, tests visible	−0.4 [−1.5,+0.6]	+0.2 [−0.8,+1.2]	−1.3 [−3.9,+1.0]	+0.4 [−1.5,+2.6]
Gap, tests hidden	−1.9 [−4.8,+0.7]	+0.5 [−2.2,+2.9]	−2.3 [−5.2,+0.3]	−0.0 [−2.0,+1.8]
Gap difference, visible − hidden	+1.5 [−1.1,+4.5]	−0.4 [−2.7,+2.2]	+0.9 [−2.1,+4.2]	+0.5 [−2.1,+3.2]

Table 11: **Cross-verification per model.** Self-authored suites: all but GPT-6 Astra. Review: all five as reviewers of every author's implementations (250–257 wrong and as many correct per author, 176 wrong and 90 correct for GPT-6 Astra; 11,430 reviews on 135 repositories). Kimi K3's review column is the last of up to three 90-minute executions.

	Fable 5	GPT-6 Astra	Kimi K3	GPT-5.6	GLM-5.2
Self-authored suites executable (%)	89.4	—	85.3	75.8	87.3
Suites passing the correct human implementation (%)	55.9 [47.9,63.7]	—	35.5 [28.7,42.5]	30.1 [23.4,37.0]	42.2 [35.0,49.6]
Suites failing wrong sibling implementations (%)	13.0 [1.0,28.9]	—	68.5 [42.5,92.7]	50.8 [23.3,78.3]	83.5 [67.0,97.5]
Repositories with a wrong sibling (sensitivity denominator)	7	—	8	10	20
As reviewer: wrong implementations accepted (%)	78.1 [73.8,82.0]	21.9 [17.1,26.8]	73.8 [68.9,78.3]	35.3 [30.0,40.8]	76.5 [71.6,81.2]
As reviewer: correct implementations rejected (%)	6.6 [4.1,9.4]	75.4 [70.4,79.9]	10.1 [7.3,13.2]	62.3 [57.1,67.3]	12.1 [8.5,16.0]
As reviewer: Youden J	0.15 [0.11,0.20]	0.03 [−0.01,0.06]	0.04 [−0.02,0.09]	0.02 [−0.03,0.07]	0.01 [−0.04,0.06]
Pairs discriminated: wrong rejected & correct accepted (%)	18.0	9.5	14.2	13.3	11.9
Review suites that separate wrong from correct (%)	15.4	14.4	11.9	15.7	7.3
Valid review suites with a known counterpart	1077	1094	1057	1094	1060

Table 12: **Attrition from verified changes to released requests.** Counts are unique requests unless stated. The two pools are repository-disjoint.

Step	Requests	Repositories
Verified changes from the history walk	25,519	1,491
in the primary pool	14,921	–
in continuity runs of at least 20	6,316	141
removed by retained-behavior eligibility	1,178	–
removed with runs below 20	240	22
remaining primary-pool requests	4,898	119
Added from the second pool (of 8,368 that pass the reference check)	1,842	74
Candidate windows: 439 chains (8,780 request occurrences)	6,740	193
After the specification audit: 397 chains	6,066	176
After the filtering rounds (Table 17): 280 chains	4,404	140

Table 13: Blind solvability campaign on the 14,921 candidate specifications (August 2026).

Outcome	Candidates
Solved $\geq 1/8$ by Claude Opus 4.8	8,695 (58.3%)
Opus 0/8, then solved $\geq 1/8$ by GPT-5.6-sol	816 (5.5%)
0/8 by both	5,145 (34.5%)
Incomplete (51 GPT runs, 214 harness E2BIG)	265 (1.8%)

Table 14: Blind-solvability class of the benchmark's 4,404 specifications (280 chains).

Class	Specifications
Solved by Claude Opus 4.8	3,183 (72.3%)
Solved only by GPT-5.6-sol	263 (6.0%)
Never blind-solved by either	886 (20.1%)
Incomplete campaign runs	72 (1.6%)

Table 15: Adequacy status of the never-solved and rewritten primary-pool specifications of the benchmark.

Status	Specifications
Hard but fair: never solved blind, no specification gap found	575
Rewritten, then solved in blind pass@4	45
Rewritten, blind validation not finished	68

Table 16: **Specification audit, from candidate windows to the re-windowed chains.**

Step	Specifications
In the 439 candidate chains	6,740
After removing one abnormal request	6,716
After a provider content filter blocked two chains	6,688
Accepted unanimously in the first wave	4,989 (74.6%)
Contested after the first wave (8,539 missing-contract findings)	1,699
Rewrites by GPT-5.6: accepted / targets	1,389 / 1,459 (95.2%)
Rewrites by Fable 5: accepted / targets	70 / 101
Discarded as still contested	62
Other requests lost when runs split (15 repositories)	560
Re-windowed: 397 chains over 176 repositories	6,066

Table 17: **Chains dropped by the filtering rounds**, from 439 candidate windows to 280 chains.

Reason	Chains
Contested specifications, provider-filtered or malformed chains, and re-windowing	42
Gold-walk, packaging, apply and window-composition failures	17
Specification hygiene, gold patches on test paths, non-reproducible repaired environments	27
Environment manipulation (§D.7)	32
Nondeterminism on an identical digest	4
Failure at the first two-container gate	20
Verifier time above 1,800 s	2
A frozen node failing under the final grader	1
Hidden tests presupposing the human implementation of an earlier segment (post hoc, six repositories)	14
Total	159

Table 18: **Benchmark-hunk reproduction.** These are probe samples, not whole-chain rates. GPT-6 Astra was not probed. Fable 5 also reproduces 21 of 145 same-repository hunks from before its cutoff (14.5%).

Model	Near-verbatim / sampled hunks	Rate (%)
Fable 5	19 / 126	15.1
GPT-5.6	5 / 406	1.2
Kimi K3	3 / 407	0.7
GLM-5.2	0 / 97	0.0

Table 19: **Resume arms per model.** D: silent continuation; E: one external "verification failed" message and one repair before request $k+1$; F: fresh conversation on the same broken state.

	Fable 5	GPT-6 Astra	Kimi K3	GPT-5.6	GLM-5.2
Starting points / control units	470 / 174	130 / 161	836 / 595	939 / 572	1451 / 1594
Control (verified state): $k+1$ own-work failure (%)	3.8	4.3	9.6	16.9	11.9
Arm D: after one untreated miss (%)	21.8 [18.3,25.8]	28.5 [21.4,36.7]	25.0 [22.2,28.1]	30.4 [27.5,33.4]	29.3 [27.0,31.7]
Multiplier, D / control	5.8 [2.2,38.0]	6.6 [2.2,24.1]	2.6 [1.6,4.8]	1.8 [1.2,2.9]	2.5 [1.8,3.4]
Difference, D − control (pp)	18.0 [8.3,28.8]	24.2 [13.1,32.1]	15.4 [7.3,23.5]	13.4 [5.4,22.2]	17.3 [11.5,23.4]
Arm F: fresh conversation, same state, F − D paired (pp)	4.1 [0.7,8.2]	—	0.4 [−1.9,2.7]	−1.2 [−5.1,2.5]	—
Arm E: repair@1 (%)	44.6 [40.1,49.1]	45.4 [37.1,54.0]	29.4 [26.4,32.6]	30.0 [27.2,33.0]	37.4 [34.9,39.9]
Arm E: $k+1$ failure, repaired states (%)	5.7	5.1	3.4	12.4	8.5
Arm E: repaired states (n)	209	59	232	282	542
Arm E: $k+1$ failure, unrepaired states (%)	22.7	28.2	25.0	30.0	31.1
Arm E: unrepaired states (n)	260	71	559	657	909
Arm E: $k+1$ failure, all states (%)	15.1 [12.2,18.7]	17.7 [12.1,25.2]	18.7 [16.1,21.6]	24.7 [22.1,27.6]	22.7 [20.6,24.9]

Table 20: **Own-work failure at requests $k+1$ to 20 by number of open defects beneath the request** (silent continuations, 140 repositories).

	Fable 5	GPT-6 Astra	Kimi K3	GPT-5.6	GLM-5.2
No open defect beneath the request (%)	8.0	8.0	12.1	12.0	10.5
One (%)	12.7	16.2	18.1	21.5	21.7
Two (%)	36.1	29.5	38.8	28.3	32.2
Three (%)	33.7	40.0	48.6	43.8	49.0
Four (%)	66.1	74.1	80.1	57.3	55.0
Five or more (%)	92.5	84.3	90.0	70.2	67.1
Units, none	893	212	994	1394	2268
Units, one	2419	672	4388	4968	7902
Units, two	781	244	1340	2270	3426
Units, three	409	75	451	726	1413
Units, four	127	27	176	391	845
Units, five or more	106	51	260	420	1276
Odds ratio per additional open defect	2.28 [1.69,3.86]	2.20 [1.69,2.79]	2.32 [1.97,3.00]	1.72 [1.49,2.05]	1.72 [1.56,1.92]

Table 21: **Labels of the 575 first-attempt silent misses on the 140 repositories** (primary judge; % of a model's misses). Benchmark-side tags are suspected from static evidence and not confirmed by re-execution.

Label	All	Fable 5	GPT-5.6	GLM-5.2	Kimi K3
Regression: pre-existing behavior broken	27.5	28	29	27	26
Missed requirement: part of the specification unimplemented	19.8	8	21	26	19
Wrong logic	10.4	11	8	10	13
Integration error: new code unreachable from the tested path	7.8	8	3	12	8
Other agent-side: wrong file, syntax, unverified assumption, knowledge gap	5.6	6	4	7	5
Benchmark-side, suspected: test stricter than the specification, broken or ambiguous	25.2	31	31	16	27
No tag	3.7	8	4	2	2
Sites	575	89	156	188	142

Table 22: **Complete-chain rates on the 280 chains (140 repositories).** Counts are chains solved; brackets are repository-clustered 95% bootstrap intervals.

Model	pass@1	[95% CI]	pass@8	[95% CI]
Fable 5	184 (65.7)	[59.1, 72.1]	223 (79.6)	[73.9, 85.1]
GPT-6 Astra	146 (52.1)	[45.0, 59.2]	171 (61.1)	[54.0, 67.9]
Kimi K3	135 (48.2)	[41.3, 55.2]	200 (71.4)	[65.2, 77.5]
GPT-5.6	123 (43.9)	[37.0, 50.7]	174 (62.1)	[55.1, 69.0]
GLM-5.2	78 (27.9)	[21.7, 34.3]	132 (47.1)	[39.9, 54.4]

Table 23: **First attempts without a verdict** on the 280 chains, and pass@1 over the first attempts that did receive a verdict.

Model	No valid record	Ended by a budget	pass@1 with a verdict
Fable 5	5	2 (wall)	184/273 (67.4)
GPT-6 Astra	0	1 (context limit)	146/279 (52.3)
Kimi K3	1	1 (tool cap)	135/278 (48.6)
GPT-5.6	0	1 (wall)	123/279 (44.1)
GLM-5.2	0	14 (wall)	78/266 (29.3)

Table 24: **First-attempt survival (reached@k) and per-request miss rate by quarter on the 280 chains.** Reached@k: share of chains whose first attempt passed state k (an attempt without a verdict counts the states it had passed, none when no record exists). Miss rate: contract failures over submissions at requests 1–5, 6–10, 11–15 and 16–20.

	Reached@k (%)					Miss rate by quarter (%)			
Model	k=5	k=10	k=15	k=20	Mean states passed	1–5	6–10	11–15	16–20
Fable 5	90.7	81.8	75.0	65.7	16.3	1.4	2.1	1.7	2.6
GPT-6 Astra	90.4	74.6	62.9	52.1	14.9	1.9	3.7	3.4	3.7
Kimi K3	82.5	66.8	59.3	48.2	13.8	3.8	4.1	2.4	3.9
GPT-5.6	87.5	70.0	57.1	43.9	14.0	2.5	4.4	3.9	5.2
GLM-5.2	72.5	55.0	40.0	27.9	11.0	5.6	5.1	6.2	6.8

Table 25: **Populations by study.** The unit and eligible subset depend on the experiment.

Study	Population and scope
Baseline and whole-chain repair	280 chains, 140 repositories.
Task-size and dependency statistics	280 chains, 140 repositories.
Verified-prefix comparison	Retained boundaries from the 140 repositories; model-specific retention.
Failed-state continuation	Eligible silent misses from the first attempts, with model-specific missing outcomes.
Blind implementation review	2,286 items; 11,430 reviews over 135 repositories; main summaries weight the four original authors equally.
Repair dose and estimated cost	Dose: 280 chains, two models; cost: modeled before the last filtering round (Appendix D.10), without GPT-6 Astra.
False-alarm study	100 stratified chains; sealed run with 100 uncensored attempts per level for both models.
Harness comparison	58 chains, 52 repositories (a 60-chain draw made before the last filtering round, minus the two chains of a repository that round excluded).
Prior-exposure probes	Hunk samples from the 140 repositories; four models, without GPT-6 Astra.
Failure anatomy and self-checks	First attempts with a finished per-request record; the self-check samples 60 silent misses and 60 accepted submissions per model (Appendix I).

Table 26: Inputs. Unit: what one trial is. Count: how the per-model count was obtained.

Benchmark	Metric (source)	Unit	n	Count
SWE-Milestone (Deng et al., 2026)	Resolve Rate (Table 2), 15 agent–model configurations	milestone	98	round(rate$\times n$), ± 1
SlopCodeBench (Orlanski et al., 2026)	strict and isolated solve (Table 1; best run per model)	checkpoint	196	round(rate$\times n$)
SWE-Chain (Lam et al., 2026)	Build+Fix resolving (Table 3), single run	chain	12	not a count
BOBA-BENCH	pass@1, attempt 1	chain	280	exact

Table 27: Pairs of models separated at $\alpha = 0.05$ (two-sided Fisher exact test on reported counts). Spread: best to worst reported rate.

Benchmark	Metric	Models	Spread (%)	All pairs	Adjacent	Top five	Leader vs. runner-up (Wilson 95%)
SWE-Milestone	Resolve Rate	15	13.4–1.3	9/105	0/14	0/10	13/98 [7.9, 21.4] vs. 12/98 [7.1, 20.2]
SlopCodeBench	strict solve	15	14.8–2.0	27/105	0/14	0/10	29/196 [10.5, 20.4] vs. 22/196 [7.5, 16.4]
SlopCodeBench	isolated solve	15	28.1–4.1	46/105	0/14	0/10	55/196 [22.2, 34.7] vs. 51/196 [20.4, 32.6]
SWE-Chain	resolving, Build+Fix	9	60.8–20.2	not testable: single run, means over 12 chains, no dispersion published; a binomial bound with 12 units separates 0/36			

Table 28: BOBA-BENCH, 280 chains, attempt 1: pass@1 with Wilson 95% intervals.

Model	pass@1	Wilson 95%
Fable 5	184/280 (65.7%)	[60.0, 71.0]
GPT-6 Astra	146/280 (52.1%)	[46.3, 57.9]
Kimi K3	135/280 (48.2%)	[42.4, 54.1]
GPT-5.6	123/280 (43.9%)	[38.2, 49.8]
GLM-5.2	78/280 (27.9%)	[22.9, 33.4]

Table 29: BOBA-BENCH pairwise comparisons on the same 280 chains. Discordant: chains passed only by the first / only by the second model. McNemar: exact two-sided p on the discordant chains. Fisher: unpaired two-sided p on the two counts. Bootstrap: repository-clustered 95% interval of the difference in percentage points.

		Diff. (pp)	Discordant	McNemar p	Fisher p	Bootstrap 95%
Fable 5	GPT-6 Astra	+13.6	54 / 16	5.9×10^{-6}	1.5×10^{-3}	[+7.5, +19.6]
Fable 5	Kimi K3	+17.5	58 / 9	6.8×10^{-10}	4.0×10^{-5}	[+12.2, +22.9]
Fable 5	GPT-5.6	+21.8	71 / 10	1.8×10^{-12}	3.1×10^{-7}	[+15.4, +28.1]
Fable 5	GLM-5.2	+37.9	112 / 6	2.1×10^{-26}	2.2×10^{-19}	[+31.6, +44.2]
GPT-6 Astra	Kimi K3	+3.9	43 / 32	0.25	0.40	[−2.8, +10.4]
GPT-6 Astra	GPT-5.6	+8.2	44 / 21	5.9×10^{-3}	0.063	[+1.9, +14.5]
GPT-6 Astra	GLM-5.2	+24.3	78 / 10	3.3×10^{-14}	6.1×10^{-9}	[+18.1, +30.5]
Kimi K3	GPT-5.6	+4.3	40 / 28	0.18	0.35	[−1.4, +9.9]
Kimi K3	GLM-5.2	+20.4	67 / 10	1.7×10^{-11}	9.8×10^{-7}	[+14.6, +26.2]
GPT-5.6	GLM-5.2	+16.1	63 / 18	5.2×10^{-7}	1.0×10^{-4}	[+10.0, +22.3]

Table 30: **What ends a first attempt**, from grader output alone (280 chains, first attempts). Each chain-ending failure is classed by its failing tests: those of the new request ($F2P_k$) and those of behavior that already existed ($P2P_k$). A regression fails at least one $P2P_k$ test. Shares are percentages of the failures with a resolved class (Resolved); the pooled row gives repository-clustered bootstrap 95% intervals. The last column gives the share of failures that break a test added or changed by an earlier request of the same chain.

	Resolved	Own tests only	Earlier tests only	Both	Regression	Earlier request broken
Fable 5	78	65.4	28.2	6.4	34.6	14.1
GPT-6 Astra	120	62.5	30.0	7.5	37.5	20.8
Kimi K3	122	70.5	23.0	6.6	29.5	13.9
GPT-5.6	144	63.2	27.8	9.0	36.8	20.8
GLM-5.2	179	71.5	21.8	6.7	28.5	13.4
Pooled	643	67.0 [60.7, 73.2]	25.7 [19.9, 31.7]	7.3 [4.1, 11.2]	33.0 [26.8, 39.3]	16.6 [12.2, 21.2]

Table 31: **Self-checks before submission**, first attempts. Each cell gives the percentage of sampled silent misses / accepted submissions (60 each per model) that show the behavior. "Green, then claimed": the last test run passed and the closing message claimed success. "Wrote test" includes scratch files. Claim rates depend on whether a model writes closing text, and the success-claim odds ratio is driven by Fable 5 (without it: 1.35, 0.71–2.70). Pooled intervals and odds ratios (OR, Mantel–Haenszel across models, silent miss vs. accepted) use a repository-clustered bootstrap.

Model	Ran tests	Repo suite	Wrote test	Saw failure	Last run green	Claimed success	Green, then claimed
Fable 5	100.0 / 98.3	73.3 / 58.3	26.7 / 5.0	56.7 / 41.7	71.7 / 68.3	65.0 / 26.7	46.7 / 20.0
GPT-6 Astra	100.0 / 100.0	91.7 / 90.0	6.7 / 0.0	90.0 / 78.3	78.3 / 65.0	98.3 / 96.7	76.7 / 61.7
Kimi K3	98.3 / 98.3	85.0 / 83.3	25.0 / 15.0	75.0 / 65.0	66.7 / 65.0	96.7 / 98.3	66.7 / 65.0
GPT-5.6	100.0 / 100.0	86.7 / 88.3	13.3 / 15.0	80.0 / 61.7	95.0 / 91.7	40.0 / 35.0	40.0 / 31.7
GLM-5.2	100.0 / 96.7	88.3 / 80.0	36.7 / 15.0	81.7 / 65.0	78.3 / 75.0	96.7 / 91.7	76.7 / 70.0
All (300 / 300)	99.7 / 98.7	85.0 / 80.0	21.7 / 10.0	76.7 / 62.3	78.0 / 73.0	79.3 / 69.7	61.3 / 49.7
95% CI, misses	98.9–100	80.4–89.3	16.7–26.6	71.2–81.9	73.1–82.8	74.1–84.1	55.7–66.9
OR (95% CI)	–	1.44 (0.90–2.29)	2.56 (1.57–4.62)	2.08 (1.41–3.07)	1.33 (0.91–1.98)	2.26 (1.40–3.71)	1.71 (1.21–2.42)

Table 32: **A silent miss that undoes accepted work** (`python-poetry/cleo`, GLM-5.2, first attempt; abridged, quotes verbatim). Request 9 is silent on a lone decorator. A grader of request 9's new test alone would accept this code. Chosen for clarity, not at random (Appendix I.3).

Request 8	Add `@app.command`, `@app.argument` and `@app.option`; a lone `@app.argument` or `@app.option` registers a command under the function's name. GLM-5.2's own script checks this, and the request passes.
Request 9	Let the three decorators stack on one function. The agent notes that a lone decorator will then stop registering a command, and decides: "that's OK because the new spec redefines their behavior." It checks only the stacked form (`ALL SPEC REQUIREMENTS VERIFIED!`) and submits: "No new test failures were introduced."
Hidden tests	Request 9's new test passes. Two request-8 tests fail: `application.has(...)` is `False` after a lone decorator. GPT-6 Astra, Kimi K3 and GPT-5.6 pass request 9 on their first attempts.

	Reject (once, at the miss)				Gate (every request): completion (%)	
		Next request fails (%) on a state that is				
Model	Repair rate (%)	repaired	unrepaired	certified	Autonomous (no supervisor)	Perfect supervisor
Fable 5	44.6	5.7	22.7	3.8	65.7	77.5
GPT-6 Astra	45.4	5.1	28.2	4.3	52.1	68.2
Kimi K3	29.4	3.4	25.0	9.6	48.2	56.1†
GPT-5.6	30.0	12.4	30.0	16.9	43.9	59.3
GLM-5.2	37.4	8.5	31.1	11.9	27.9	41.4

Noisy supervisor: completion (%), 100-chain subset per model

Model	Autonomous	p=0	p=0.1	p=0.2	p=0.3	p=0.4	p=0.5
GPT-6 Astra	49	64	64	58	61	53	53
GPT-5.6	44	60	58	54	46	49	47

Table 33: **What one bit from outside recovers: summary of Section 5** (280 chains). *Reject*: the message once, at a silent miss, with one repair round; *Gate*: the same after every failed submission of a chain. The certified column is the reference of a state that passed its contract. The lower block reports the separate 100-chain *Noisy* study: p is the false-alarm rate and p=0 a perfect supervisor. †Lower bound (Appendix K).

Table 34: **Name and Insist** (280 chains). Repair rate of the starting-point miss after r rounds; increments per round; calls until the miss passes among repaired misses and calls spent among misses never repaired (Insist arm). Wilson intervals for rates, repository-clustered bootstrap for the paired contrast.

	GPT-5.6	Kimi K3
Starting points, Reject / Name / Insist (repositories)	939 / 939 / 939 (100)	834 / 827 / 821 (96)
Reject, repair rate after one round (%)	30.0 [27.2, 33.0]	29.5 [26.5, 32.7]
Name, repair rate after one round (%)	46.4 [43.3, 49.6]	46.8 [43.4, 50.2]
Name − Reject, paired (pp)	16.4 [11.0, 22.0]	17.1 [11.2, 22.8]
Insist, repair rate after round 1, 2, …, 8 (%)	30.0, 38.1, 44.3, 47.6, 49.8, 52.2, 53.0, 53.9	31.1, 42.3, 48.2, 52.6, 54.1, 56.0, 57.6, 58.7
Increment of rounds 2, …, 8 (pp)	8.1, 6.2, 3.3, 2.2, 2.3, 0.9, 0.9	11.2, 6.0, 4.4, 1.5, 2.0, 1.6, 1.1
Round 8, Wilson interval (%)	53.9 [50.7, 57.1]	58.7 [55.3, 62.0]
Calls until pass, repaired misses, median	16.5	19
Calls spent, misses never repaired, median (mean)	78 (87.8)	73 (75.5)

Table 35: **Gate** (280 chains). Completion with no supervisor (first attempt) and with a perfect supervisor (one repair round at every failed request); Wilson intervals over 280 chains.

	Fable 5	GPT-6 Astra	Kimi K3	GPT-5.6	GLM-5.2
Autonomous: first attempts completing 20 (of 280)	184	146	135	123	78
Failed first attempts audited	96	134	145	157	202
Rescued by verification and repair	33	45	22	43	38
Perfect supervisor: completion (%)	77.5 [72.3, 82.0]	68.2 [62.5, 73.4]	56.1 [50.2, 61.8]	59.3 [53.4, 64.9]	41.4 [35.8, 47.3]
Gain (pp; relative)	11.8 (18%)	16.1 (31%)	7.9 (16%)	15.4 (35%)	13.6 (49%)
Unresolved continuations (counted as failures)	0	0	18	0	0
Upper bound if all unresolved had completed (%)	—	—	62.5	—	—

Table 36: **Noisy supervisor** (sealed run; 100 chains per level, 0 censored). Completion against the false-alarm rate p, its paired difference from the perfect supervisor on the same chains, and the fate of the phantom rounds. Brackets: repository-clustered bootstrap 95% intervals (2,000 replicates).

	p=0.1	p=0.2	p=0.3	p=0.4	p=0.5
GPT-5.6-sol — autonomous 44 [35, 54]; perfect supervisor 60 [50, 70]					
Completion (%)	58 [49, 67]	54 [44, 64]	46 [37, 56]	49 [39, 59]	47 [37, 57]
Difference from perfect supervisor, paired (pp)	−2 [−9, 5]	−6 [−13, 1]	−14 [−23, −5]	−11 [−19, −2]	−13 [−21, −5]
Chains ended by a false alarm (%)	3	2	8	7	10
Phantom rounds (n); broken (%); unchanged (%)	147; 2.0; 2.0	298; 0.7; 1.7	416; 1.9; 0.2	581; 1.2; 1.2	701; 1.4; 2.0
Calls per phantom round, median	8	8	7.5	8	8
Phantom rounds' share of list-price spend (%)	9	17	22	26	32
True alarms repaired (%)	33 (19/58)	34 (23/67)	27 (17/63)	24 (14/58)	25 (14/57)
GPT-6 Astra — autonomous 49 [40, 59]; perfect supervisor 64 [54, 73]					
Completion (%)	64 [55, 73]	58 [48, 68]	61 [51, 71]	53 [44, 63]	53 [42, 62]
Difference from perfect supervisor, paired (pp)	0 [−7, 7]	−6 [−13, 1]	−3 [−10, 4]	−11 [−19, −3]	−11 [−19, −2]
Chains ended by a false alarm (%)	6	6	7	8	12
Phantom rounds (n); broken (%); unchanged (%)	157; 3.8; 0.0	311; 1.9; 0.3	463; 1.5; 0.9	593; 1.3; 0.8	771; 1.6; 0.6
Calls per phantom round, median	6	6	6	5.5	6
Phantom rounds' share of list-price spend (%)	9	17	22	27	32
True alarms repaired (%)	47 (27/57)	36 (20/56)	40 (21/53)	33 (19/58)	35 (19/54)

Model	Scaffold	pass@1 [95% CI]
GPT-5.6-sol	mini-swe-agent	24/58 [29.2, 55.4]
	Codex CLI	29/58 [36.7, 64.8]
	OpenHands SDK	28/58 [35.0, 62.7]
Kimi K3	mini-swe-agent	29/58 [36.2, 64.8]
	OpenHands SDK	25/58 [29.6, 57.1]
	pi	27/58 [32.3, 61.7]

Table 37: **Scaffold sensitivity.** First attempts on a stratified 58-task subset (52 repositories; a 60-task draw made before the last filtering round, minus the two chains of a repository that round excluded) under agents installed inside the task container, same hidden grader and budgets; repository-clustered 95% bootstrap intervals in percent.