BENCHMARKS AS SOFTWARE: TOWARD CONTINUOUS BENCHMARKS FROM TERMINAL-BENCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are among the most economically consequential deployments of LLMs to date, making reliable evaluation increasingly important. Yet even benchmarks that are carefully validated at release can become stale. As models improve, they can discover valid solutions that existing verifiers were not designed to recognize, while changes to evaluation infrastructure can introduce new sources of error. We call the resulting accumulation of defects *benchmark maintenance debt*: a growing gap between benchmark scores and the capabilities they intend to measure.

We introduce a Continuous-Validation Workflow for agentic benchmarks that combines community reports, task artifacts, agent traces, automated oracle checks, and human-gated repairs. We apply this workflow to Terminal-Bench 2.0, a benchmark that underwent extensive human and LM-assisted review and red-teaming before release. Our audit identifies issues in roughly 30% of the benchmark, and we release the resulting repairs as Terminal-Bench 2.1. These repairs change evaluation results: across agents, accuracy shifts by as much as 12 percentage points and leaderboard positions move by up to three ranks, even as the overall ranking remains highly correlated (Spearman $\rho = 0.958$). We also show the workflow generalizes to tasks in other agentic benchmarks.

Beyond accuracy, we evaluate how resource allocation shapes benchmark conclusions. We show that fixed wall-clock and output-token budgets can materially change apparent agent performance: some large harness gaps at the default timeout shrink substantially when agents are given more time. These results show that benchmark maintenance is not only about repairing task artifacts; resource budgets are part of the measurement specification and must be validated alongside the tasks themselves. More broadly, benchmark maintenance debt is not just accumulated implementation bugs: our results show that as models improve, both evaluation assumptions and resource settings may need to be revalidated.

1 INTRODUCTION

Agentic benchmark scores shape product claims and research investment. But agentic benchmarks are not static datasets: over time, these systems accumulate defects that decouple a model's score from the capability the benchmark is meant to measure. In turn *benchmark maintenance debt*, the backlog of task-side defects that must be repaired or retired for benchmarks to remain valid, is accumulated along the following axes:

- **Fidelity debt**: benchmark designers can encode assumptions about how agents will solve a task, which break when agents discover shortcuts such as hardcoded solutions, substring-matching exploits, or valid solution paths the designer did not anticipate.
- **Drift debt**: Tasks that depend on external state (packages, APIs, websites) can break when that state changes after release, causing failures unrelated to agent capability.

For agentic benchmarks, validity is therefore not a one-time certificate, it is a maintenance problem. The problem can be illustrated with Terminal-Bench 2.0 (Merrill et al., 2026), an agentic benchmark reported in model cards from OpenAI (OpenAI, 2024a), Anthropic (Anthropic, 2026a), Cursor (Cursor Research Team, 2026), Qwen (Qwen Team, 2026c), GLM (Zeng et al., 2026), and Google (Google DeepMind, 2026b), among others. Terminal-Bench 2.0 tasks were manually reviewed, LM-audited, and exploit-tested before release to ensure quality following best practices (Zhu et al., 2025). However,

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

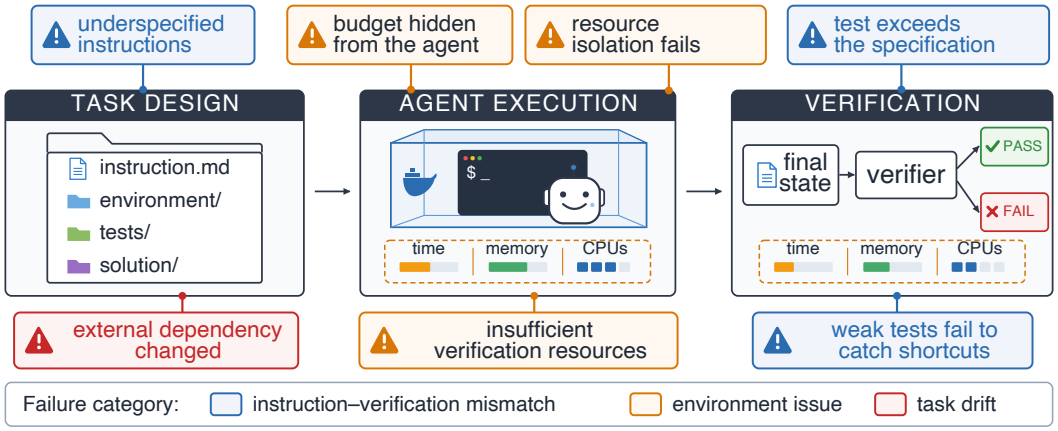

Figure 1: **Failure surfaces along the agentic-task lifecycle**. From authoring to execution and verification, each stage can admit a failure that contributes to benchmark maintenance debt. Colors mark the three failure categories (blue = instruction–verification mismatch, orange = environment issues, red = task drift).

release-time validation is not enough: within months, the benchmark had accumulated over 50 open GitHub issues tagged as task defects (Terminal-Bench Team, 2026a;b; Zeng et al., 2026), spanning different assumptions made across the full task stack (Fig. 1). Furthermore, many of these issues came with proposed repairs of varying quality, highlighting the lack of community standards for repairing benchmarks. Some proposals addressed the underlying validity failure by clarifying the task or repairing the oracle; others only patched the observed symptom, for example by relaxing a regex or broadening an acceptance condition. Diagnosing the root cause was further complicated by infrastructure variance: differences in resource allocation, environment configuration, and model and sandbox providers meant that the same task could behave differently across evaluation setups (Anthropic, 2026; OpenAI, 2025b).

Existing responses address parts of benchmark maintenance debt. One response is to freeze the environment and isolate it from the web. Freezing improves reproducibility, but at a cost to fidelity: the benchmark becomes stable by becoming less like the live setting it was meant to model. Another response is to filter rather than repair, using only a validated subset of the benchmark (OpenAI, 2024b). Filtering improves precision, but agentic tasks are expensive software artifacts and removing them reduces coverage and can change the benchmark's difficulty distribution. A third response is to refresh the benchmark continuously (Jain et al., 2025). Refreshing addresses contamination (such as from reward hacking) and staleness, but it does not answer the post-release maintenance question: when defects are discovered, which tasks can be repaired rather than filtered, what validation is sufficient for each repair, and how much do these decisions change reported accuracy and cost?

We address benchmark maintenance debt in three steps:

- **A continuous validation workflow.** We show that benchmark validation can itself be continuous: modern agentic tools make it practical to combine community reports, public execution traces, agent-assisted triage, and human adjudication into a maintainable workflow for identifying, repairing, retiring, and documenting benchmark issues (Sec. 4).
- **An updated benchmark, Terminal Bench 2.1.** We apply this workflow to Terminal-Bench 2.0 and release Terminal-Bench 2.1. The workflow surfaces issues in roughly one third of the benchmark. We categorize these issues into three recurring sources of maintenance debt: *environment issues*, *instruction–verification mismatches*, and *task drift* (Sec. 5).
- **The effects of benchmark repair.** We re-evaluate 19 agents on Terminal-Bench 2.0 and Terminal-Bench 2.1 and show that repair can change reported accuracy and efficiency conclusions without fully overturning the leaderboard. Agent rankings remain highly correlated, but individual scores move by nontrivial margins (Sec. 6). Even with feasibility restored, we highlight the changes in performance due to resource allocation (e.g. >50% of failed trials are due to timeouts, and up to ∼68% for some harnesses) (Fig. 3b).

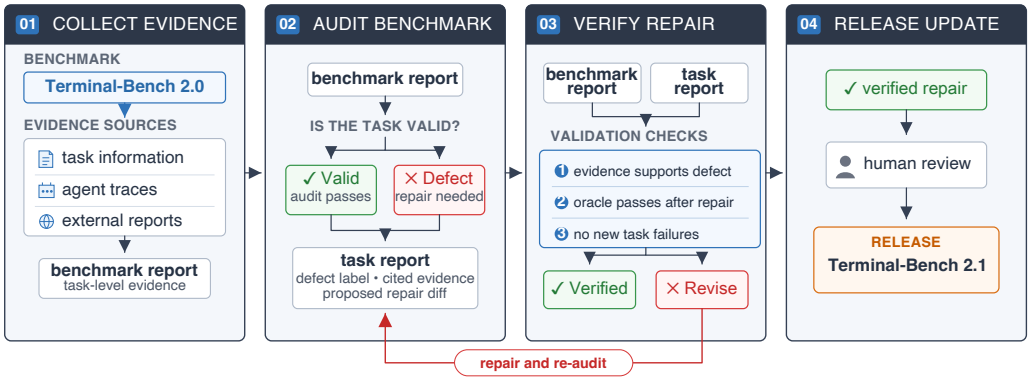

Figure 2: **Agentic pipeline for continuous benchmark validation.** An evidence collector agent gathers complementary evidence streams: static task files, prior agent traces, and external reports. An auditor agent applies audit principles for specification fidelity, tests, resources, and risk, then drafts a cited repair, which is checked against evidence while `/oracle` rebuilds the task and runs the reference solution. The repair workflow can open a draft PR with evidence; maintainer review gates the verified release.

2 Related Work

Benchmark auditing. Prior work has studied how benchmark conclusions change when the evaluation set is revised. Recht et al. (2019) resampled ImageNet and CIFAR test sets and used rank correlation between the original and new test sets to evaluate whether a benchmark measures a stable construct. We adopt the same rank-stability protocol, but apply it to a different failure mode: task-side defects in an executable agent benchmark.

A second response is to validate and filter benchmark instances. SWE-Bench Verified (OpenAI, 2024b) contracted professional developers to audit SWE-Bench (Jimenez et al., 2024) and released a conservatively filtered subset. WebArena Verified (ServiceNow Research, 2026) follows a similar validated-subset approach for web-agent tasks. Related audits identify solution leakage, weak test suites, and task/outcome-validity failures in agentic benchmarks (Aleithan et al., 2024; Yu et al., 2025; Zhu et al., 2025). These approaches typically treat validation as a one-shot curation pass and reduce the benchmark rather than repairing it.

A third response is to time-window or refresh benchmarks. LiveCodeBench (Jain et al., 2025) and LiveBench (White et al., 2025) continuously add new problems to reduce contamination and benchmark staleness. This approach is closest in spirit to continuous validation, but instead we focus on repairing rather than filtering.

Auditing tooling. Benchmark auditing is facilitated by a growing stack of operational tools. The Agentic Benchmark Checklist (ABC) (Zhu et al., 2025) provides rubrics for auditing outcome validity and task validity. Tools such as Docent (Meng et al., 2025) and LangSmith LangChain (2026) make large-scale inspection tractable by ingesting benchmark trajectories for review. Complementary adversarial tools audit benchmark hackability by searching for ways that agents can pass tests without satisfying the intended task. These tools address different parts of the audit pipeline. Rubrics specify the evaluation criteria; ingestion tools collect evidence; red-teamers identify candidate defects; and human adjudication resolves ambiguous cases. We combine these ingredients into a continuous validation workflow (Sec. 4).

Agentic evaluation. Executable agent benchmarks do not evaluate a model in isolation: scores also depend on the agent harness, the execution environment, and the resource budget. Harness effects are well documented (Yang et al., 2024; Wang et al., 2025). Merrill et al. (2026) found that, on average, the choice of harness matters less than the choice of model. Environment effects have received less attention: Anthropic (2026) shows that infrastructure configuration alone can shift agentic coding scores. We extend this line of work in two ways: averages can hide large within-vendor harness gaps ($\sim$20 pp for some models), and that these gaps largely reflect speed under the default time budget.

3 PROBLEM SETUP

Every task has a failure surface. A task is a timed exam with the following components:

- **Instruction:** the natural-language problem description shown to the agent.
- **Environment:** the Docker image with pre-installed software, network rules, and any provided files.
- **Oracle:** the known-correct solution.
- **Test:** the grading code that runs after the agent stops.
- **Resources:** CPU, memory, and wall-clock budgets.

An *agent* is a model with access to tools. The *agent harness* is the system that gives a model the tools it needs to interact with its environment. A *trial* is one independent attempt at a task by an agent.

For agent a working on task t with $n_{a,t}$ trials, $\hat{p}_{a,t}$ is the mean reward. The coverage, or *pass@k*, estimates the probability that at least one of the trials succeeds. The coverage is the unbiased pass@k estimator from Chen et al. 2021 throughout.

Datasets. We operate with two datasets: the *evidence data* and the *evaluation data*. The evidence data is the external evidence gathered on the benchmark, including trial data from the leaderboard, used to identify and fix defective tasks and release Terminal Bench 2.1. The evaluation data is a fresh set of TB2.0 and TB2.1 trials, run for 19 agents, used to measure the effect of the fixes.

To isolate the role of efficiency, we run an additional condition on TB2.0 and TB2.1 that deliberately deviates from the benchmark guidelines: we increase the resources allocated, giving agents more time to complete each task. Owing to cost, this condition is run on a subset of the 19 agents.

4 BENCHMARK AUDITING WORKFLOW

Our goal is to audit agentic benchmarks for task-side defects and to produce candidate repairs that maintainers can review. We have at our disposal three evidence streams: the task's specification, prior execution traces, and community reports from the web. We want this evidence compiled into a task-level audit report: a defect label, supporting evidence, and a proposed fix when one is available. We define three agent skills for collecting evidence, auditing, and verifying the final report.

Audit rubric. A central requirement of the workflow is a fixed rubric. Without one, an agent can flag suspicious behavior, but its behavior may not be consistent or easy for a human to parse. Previous work has developed rubrics to audit benchmarks (Zhu et al., 2025). We build on this idea, but adapt it for a maintenance workflow. In this setting, we found it more useful to ask for a small number of action-oriented defect labels tied to likely repairs, rather than broad free-form validity judgments. The auditor therefore assigns a primary task-side defect category and records secondary labels when applicable. This makes reports comparable across agents and gives maintainers a concrete review target. We ask auditors to classify each task using the following primary *defect categories*:

- **No defect found:** the task is well specified, the oracle is accepted, the verifier checks the intended behavior, and the stated resource budget is sufficient.
- **Instruction–verification mismatch:** the task instruction and the verifier disagree. This includes underspecified instructions whose hidden constraints appear only in tests, as well as overspecified tests that reject behavior satisfying the task as written.
- **Task drift:** the task changed after release because of mutable external state, such as package versions, network resources, public APIs, or web content.
- **Environment issues:** the task is solvable in principle, but the provided environment and resources (CPU, memory, storage, or system/network configuration) prevent the oracle or verifier from running reliably.

Audit workflow The `audit` workflow is illustrated in Fig. 2. For each task, the *evidence* agent produces a structured report containing information on the benchmark, the task specification, trace information, and community reports. The role of this agent is to organize available evidence for a downstream auditor, while keeping observation separate from adjudication.

An *auditor* applies the rubric to each task. The auditor checks whether the task is well specified, solvable, verifiable, and whether the verifier accepts valid solutions and rejects invalid ones. The

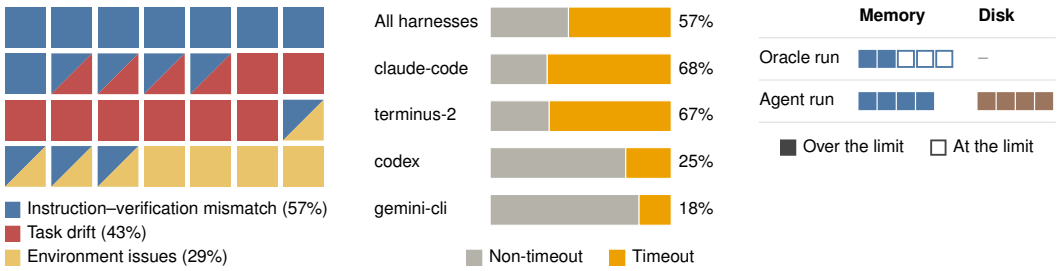

Figure 3: **Three views of where task errors come from.** (a) Prevalence of defect categories in 28 tasks from Terminal Bench 2.0; a task can have more than one. (b) Prevalence of timeouts among failed Terminal Bench 2.1 trials at the default time budget, by harness (the set of agents varies by harness). (c) Terminal Bench 2.0 tasks whose memory or disk limit was too tight; each square is one task. Not all resource issues are captured by the oracle runs.

rubric here follows the ABC guidelines Zhu et al. (2025), and it produces a task-level report containing a primary defect label, a short justification, the location of the defect, and a proposed repair.

A *verifier* reviews the evidence and the auditor's label. We include this stage because auditing alone did not capture task drift reliably in the pilot runs, as we describe next. The verifier agent is asked to run the oracle, challenge the proposed label, identify missing evidence, and check whether the proposed fix addresses the underlying defect. The full breakdown is in Appx. A.

Repair conventions. Once a defect is identified, the repair is not always mechanical, but involves human judgment (Zhu et al., 2025). We follow these conventions for each defect category:

- **Instruction–verification mismatch**: any solution satisfying the task as written is accepted, and invalid solutions are rejected.
- **Task drift**: make the dependency reproducible where possible and remove any dependency that prevents the oracle from running reliably.

Environment Issues. We set a *feasibility floor*: a known-correct oracle solution and the verifier should run reliably within the benchmark budget. For resource-related failures, we adjust only the binding resource axis (wall-clock time, memory, or CPU count) to the factor closest to the original proposal or at least $1.5\times$ the observed requirement of the oracle solution and verifier.

This convention serves two purposes. First, it prevents false negatives caused by infrastructure noise: a correct solution should not fail because the verifier or oracle only barely fits within the budget, but we should leave enough slack for the problem to be solved. Second, setting the resource sets the task difficulty, and here we leave those as close to what the original task contributors intended.

An alternative would be to tune resource budgets per task. However, doing so would require repeated sweeps over any resource and would introduce an additional subjective design choice into the process. While the feasibility rule is conservative, it is an auditable convention.

Cross-cutting conventions. When tasks depend on live APIs, the difficulty can change after the audit. When a repair changes task difficulty, we document the change rather than tuning the task back to match its previous pass rate. Repairs are adjudicated in the open, and keyed to the defect category they address.

Workflow Evaluation We evaluate the audit workflow in the setting it was designed for: post-release benchmark maintenance. We used the workflow during Terminal-Bench 2.0 maintenance to surface candidate task defects, attach evidence, and propose repairs for maintainer review. The accepted repairs were then shipped as Terminal-Bench 2.1.

We evaluate the workflow along three dimensions. First, we ask whether the workflow is agent-agnostic: under the same rubric, do different auditors assign similar defect labels? Second, we ask whether this agreement persists when the workflow is instantiated with lower-cost auditor models. Third, we evaluate the workflow as a maintainer-in-the-loop triage system by comparing its flags against a human-adjudicated set of repairs.

As we described previously, the agents assign a task to a defect category. The defect categories were selected based on results from a 15-task pilot subset of Terminal-Bench 2.0 and approximately 1,500 TB2.0 execution trials. The pilot fixed only the category definitions, not per-task labels.

To evaluate whether the workflow is agent-agnostic, we ran it with two classes of auditors. To prevent leakage from the community reports the workflow is designed to consume, all evaluation runs disable the auditors' web-search tools. Managed coding assistants, Claude Code (Opus 4.6) and Codex (GPT-5.4), agree on the primary defect category, including *no defect*, at $\kappa = 0.80$ over all 89 TB2.0 tasks.

Lower-cost API auditors agree on the binary triage decision (flagged vs. clean) at $\kappa = 0.55$–0.77 over the same 89 tasks (Table 1), but less on attribution: per-category agreement (one-vs-rest, since a task can carry multiple labels) is moderate for instruction–verification mismatch ($\kappa = 0.60$) and environment issues ($\kappa = 0.59$), and low for task drift ($\kappa = 0.21$–0.41). We therefore rely on the deterministic oracle check rather than auditor labels to detect drift: every drift defect modifies the environment or solution code, so rerunning the reference solution surfaces it.

These results suggest that capable auditors are largely agent-agnostic in both triage and attribution, while lower-cost auditors are reliable for triage but require the oracle check and maintainer review for attribution. Agreement measures consistency rather than correctness. An extended breakdown appears in Appx. A.

Against the final TB2.1 release, the workflow reaches 57.1% precision and 71.4% recall. Because 14 of the 28 repairs were first surfaced by the workflow itself, recall partly measures agreement with the release rather than independent detection; the workflow is best viewed as a triage tool whose flags maintainer review then filters.

5 From Terminal Bench 2.0 to 2.1

Applying the audit workflow to Terminal-Bench 2.0 surfaced issues in 28 tasks, roughly a third of the benchmark. These consolidate defects found in prior audits, namely the 11 task repairs from Terminal-Bench Verified Z.ai (2026) and 5 further tasks addressed across various public issues and pull requests Terminal-Bench Team (2026a;b), together with 14 additional tasks that the audit newly identified as needing repair. Two tasks can appear in more than one source, giving 28 in total.

Fig. 3 shows the breakdown of the task defects in the 28 tasks that needed repair. The most common failure mode was instruction–verification mismatch (present in 57% of the tasks). For example, on the task `query-optimize`, the instruction said not to modify the database, but the test checked a hash; an agent that touched and reverted the file failed even though the database was logically identical (Aleithan et al., 2024; Yu et al., 2025).

The second most common defect was task drift (43% of tasks). Tasks that depend on external state, such as package versions, public APIs, and web content, break when that state changes after release. We repair drift by removing the dependency where possible and pinning it otherwise (See Appx. A).

The third category, environment issues (29% of tasks), comes from resource budgets. In TB2.0, task contributors set CPU, memory, and time budgets for each task based on their own resource use. These estimates were too tight in several cases (Fig. 3c): the oracle solution, the verifier, or the agent ran out of memory or disk, so trials could fail because of resource limits rather than incorrect solutions. Execution backends (Docker, Daytona) also differ in how they isolate filesystems, enforce memory limits, and expose networks between agent and verifier. The audit raises each budget so the oracle runs with at least $1.5\times$ slack on the binding axis, restoring a feasibility floor.

An audit of this size is dangerous: if agent performance reorders the leaderboard, every prior claim on TB2.0 is suspect, ours included. We test this directly in the next sections.

6 Results

The performance of an agent is a product of the model, harness and resources Yang et al. (2024); Kapoor et al. (2025); Merrill et al. (2026). While we generally compare model and harness, here we show the impact of resources as well. We use TB2.1 to ask which conclusions are stable under changes to this measurement stack. First, we compare TB2.0 and TB2.1 at the default protocol to

test whether repairs change accuracy and rankings. Second, we vary the time budget to test whether repairs change the efficiency frontier: which agents solve tasks, and how much budget they need to do so. Third, we ask how much of a fixed-budget harness gap is explained by speed under the timeout. Scalar leaderboards hide these distinctions because they collapse benchmark version, harness, and resource budget into one number.

6.1 The audit raises accuracy and changes close rankings

We evaluate 19 agents from multiple providers and compare the accuracy in both TB 2.0 and TB 2.1, as illustrated in Fig. 4. The correlation is high (Spearman $\rho = 0.958$), with an OLS fit of $\text{TB2.1} = 2.6 + 1.03 \cdot \text{TB2.0}$. The lift is uneven: Claude Code + Claude Opus 4.6 lifts $+7.3$ pp above the fit (rank $8 \rightarrow 5$), while Codex + GPT 5.4 sits 7.8 pp below the fit. Negative-residual agents concentrate below 50% on TB 2.0 accuracy, suggesting that weaker agents benefit less from the repairs. This aligns with the distribution of difficulty in the fixed tasks: 19 of the 28 repaired tasks were hard, 7 medium, and 2 easy.

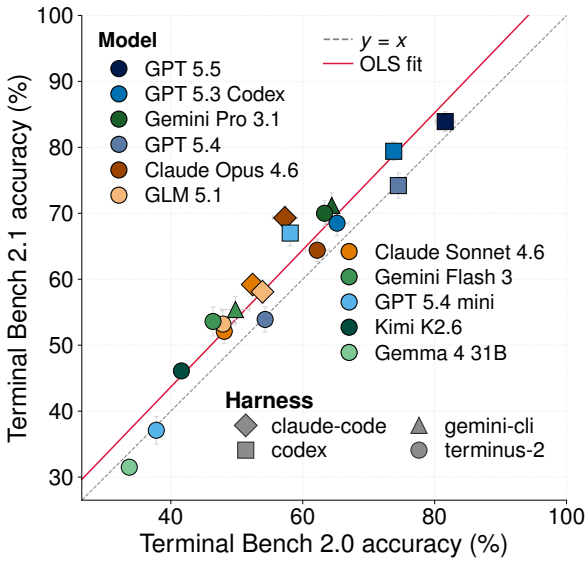

Figure 4: **Terminal Bench 2.1 changes accuracy unevenly across agents.** Spearman $\rho = 0.958$ with a task-bootstrap 95% CI of $[0.881, 0.979]$ across 19 agents.

Verifier fixes account for the largest score movements. In TB 2.1, five tasks increased from pass rates of at most 15% to over 50% after repairs centered on the verifier, accounting for roughly 62.7% of the total absolute pass-rate change on the modified-task subset (full breakdown in Appx. C). These tasks fall under instruction–verification mismatch: agents were often producing valid solutions, but TB 2.0 failed to recognize them. The remaining 23 modified tasks contribute smaller changes in both directions. In some cases, the audit made tasks stricter by removing false-positive credit or tightening the intended specification, lowering TB 2.1 pass rates relative to TB 2.0.

6.2 The audit shifts the efficiency frontier

Setting resources for executable agent benchmarks is non-trivial. A task score depends not only on whether an agent can solve the task, but also on whether it can solve it within the allocated resources. Fig. 3b shows that over 50% of failed trials are due to timeouts rather than incorrect answers, ranging from 68% under `claude-code` to 18% under `gemini-cli`; for the weakest agent–model pairs, timeouts dominate (Appx. C.2).

We therefore evaluate the test-time scaling properties of these models. For each agent–model pair, we increase the wall-clock budget up to $32\times$ the original task timeout and measure how many tasks are solved as the budget increases. This larger budget is an upper bound: agents can exit once they finish, so they do not need to consume the full allocation. At this budget, timeout errors disappear for the agent–model pairs considered here, letting us ask which failures remain once we remove time as a binding constraint.

Fig. 5 shows coverage for six agent–model pairs: five use Terminus-2, while GLM 5.1 with Claude Code is included as a reference point because it lies on the efficient frontier. The solid curves show TB2.1 and the dashed curves show TB2.0. Each curve should be read in two ways. Its left–right position shows efficiency: a curve that rises earlier corresponds to an agent that solves tasks with less time or fewer output tokens. Its final height shows coverage at the largest budget: if the curve flattens below 1.0, then giving more budget does not recover the remaining tasks.

First, agents differ substantially in efficiency. GLM 5.1 reaches high coverage at much smaller budgets than the other agents, so its curve sits farther to the left. GPT 5.4 reaches higher final

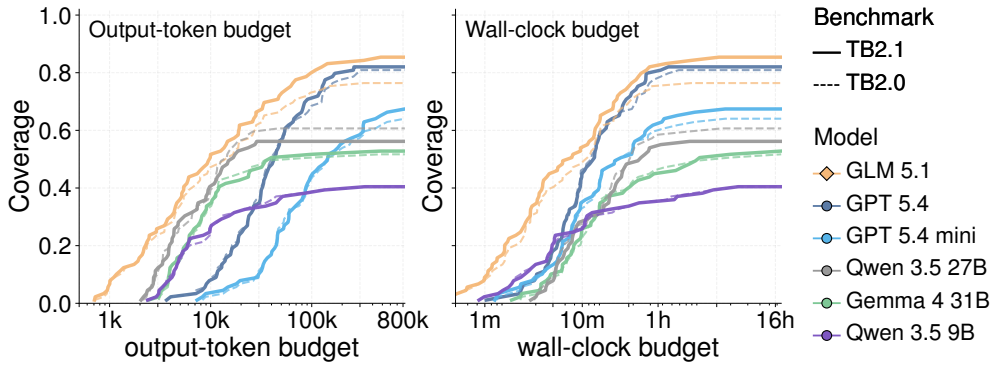

Figure 5: **Cost-aware effectiveness shifts across the audit.** Coverage improves as resource budgets increase.

coverage, but requires more budget to get there. The smaller open models, including Qwen 3.5 9B, Qwen 3.5 27B, and Gemma 4 31B, solve many tasks as the budget increases, but their curves flatten at lower coverage. Since the wall-clock sweep removes timeout errors in these runs, the remaining failures are not timeout artifacts, but tasks the models cannot solve.

Second, the audit changes these curves unevenly. For GLM 5.1, TB2.1 credits substantially more solutions than TB2.0, shifting the curve upward. For the Qwen pairs, by contrast, the TB2.0 and TB2.1 curves are nearly unchanged. Thus, benchmark repair does not simply add a constant score increase. It changes which solutions are credited, and therefore can change which agents appear efficient at a given budget.

Harness gaps are speed gaps. Prior work has shown that harnesses and agent-computer interfaces affect agent performance (Yang et al., 2024; Kapoor et al., 2025; Merrill et al., 2026). We ask: how much of a fixed-budget harness gap disappears when the slower harness is moved along the efficiency frontier?

Fig. 6 uses within-vendor harness swaps as a diagnostic. At the default time, Codex gives much higher scores than Terminus-2 for OpenAI models. Harness choice moves GPT 5.4's accuracy by ∼20 pp, rivaling between-model gaps. However, most of the harness gap is recovered by raising the time, so at the default budget the harness gap largely reflects speed rather than capability.

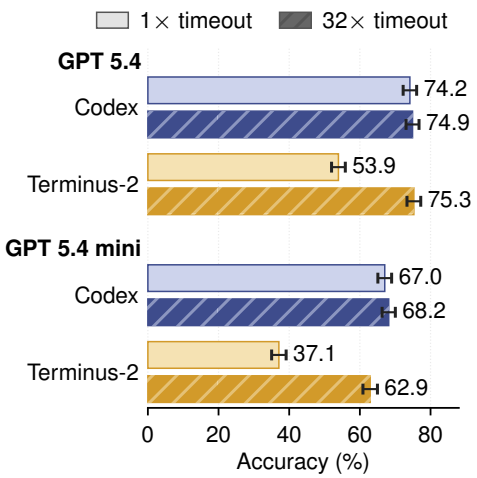

Figure 6: **The harness gap is mostly a time-budget effect.** TB2.1 accuracy for Codex and Terminus-2 at the standard allocated time and with timeouts extended to 32×. Extending the budget closes most of the Codex–Terminus-2 gap.

The results here show that a fixed resource budget is an arbitrary operating point and that the choice is not innocuous: it can change efficiency conclusions for individual models. This supports setting a principled floor and testing sensitivity to the budget, rather than calibrating to any one choice.

7 External validity: Auditing to build Harbor-Index v0.1

To test whether our workflow generalizes beyond Terminal-Bench, we apply it to Harbor-Index v0.1, a unified benchmark constructed from an initial pool of 54 candidate benchmarks (Shi et al., 2026). The final benchmark contains 80 tasks drawn from 29 source benchmarks spanning diverse domains, environments, and evaluation procedures. Unlike Terminal-Bench, many of these tasks were

reimplemented from existing benchmarks into a shared format, creating an additional opportunity for defects to be introduced in instructions, runtime environments, and grading logic.

We ran the continuous-validation workflow on 34 tasks(43%) of Harbor-Index, covering tasks from GSO, GAIA2, Lab-Bench, GPQA Diamond, OmniMath, and other constituent benchmarks. The workflow flagged 14 tasks for further inspection. Manual adjudication confirmed defects in 12 tasks (precision of 86%). The confirmed defects included failures in task infrastructure, verifier logic, test correctness, and dependencies on mutable external state.

The workflow is particularly effective at finding instruction verification issues. For example, several GAIA2 tasks failed the oracle check because an auxiliary grading service was invoked under the wrong condition, causing correct reference solutions to be rejected. For GSO, some instructions omit a condition that the grader requires for success. The workflow also identified tasks affected by task drift.

Manual adjudication reveals two kinds of failures that the workflow misses. First, it misses three task-specific defects: a Lab-Bench task with an incorrect reference answer, a QCircuitBench task whose score is nondeterministic across repeated runs, and an HLE task whose instructions admit two plausible correct answers. Second, we identify two systematic grader bugs affecting 13 tasks in total. The GSO grader executed the agent-produced program twice, and the GAIA2 grader mishandled special characters. These cases expose a limitation of task-level auditing: some failures require repeated execution or inspection of grading logic shared across multiple tasks.

Overall, the Harbor-Index audit shows that the workflow transfers beyond Terminal-Bench and can identify defects introduced across different stages of benchmark construction. The most common failures it surfaces are instruction–verification mismatches, errors in grading logic, and task drift. These are the kinds of failures that are difficult to catch through static inspection alone because the task may appear valid until the oracle or an agent is actually executed.

At the same time, the missed cases show that no single automated audit is sufficient. Some defects only appear under repeated execution, such as nondeterministic grading behavior, while others require domain knowledge to determine whether the problem lies in the task specification, reference answer, or verifier. The shared grader failures also show that task-level auditing can sometimes miss systematic issues that affect multiple tasks at once. We therefore view the workflow as a maintainer-in-the-loop validation system: it narrows the set of suspicious tasks, provides concrete evidence for likely failures, and makes repair more targeted, while human review and execution-based testing remain necessary for ambiguous or cross-cutting defects.

8 Discussion

Continuous validation. The audit pipeline surfaced 28 task-side defects in TB2.0, including 12 cases of post-release drift. Drift is by definition a recurring failure mode: more dependencies will rot between this paper's submission and its publication. This is the case for version control rather than one-shot validation. Furthermore while we didn't see reward hacking issues in this study, this behavior can contaminate tasks as well.

Numbered, dated revisions with a `CHANGELOG.md` fit this setting better. LiveCodeBench (Jain et al., 2025) already operates this way for contamination control; the same discipline applies to task-side defects.

Performance preservation under audit. Across 19 agents, the maximum difference in rank ordering is 3. Agent-level score shifts have median 5.3 pp (IQR 2.3–6.8). The largest gains concentrate on instruction–verification mismatches, the most prevalent defect category. The audit also shows the sensitivity of efficiency conclusions to different design choices, and how setting a feasibility floor is an affordable fix.

Overall, this work provides (i) an audit methodology that surfaces and repairs benchmark defects (ii) the TB2.1 artifact, demonstrating that audit-driven repairs can preserve agent rank ordering while shifting absolute scores; and (iii) evidence that, on TB2.1, both task definitions and resource budgets shape published scores in ways current reporting practice does not control for. Finally, because this pipeline can run continuously, Terminal Bench 2.1 is a checkpoint rather than a terminal state.

AI USE STATEMENT

In this work, we used generative AI tools to help develop theoretical models or conceptual frameworks, propose or refine hypotheses, implement methods, design or provide feedback on research methodology or experiments, and clean and reformat datasets.

We have not used generative AI tools to formulate mathematical claims, generate synthetic data sets, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, assist with translation, support qualitative and thematic data analysis, interpret results.

Additionally, we used generative AI tools to create or modify scientific figures or images, create or edit software code. We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work audits and repairs a public software benchmark. The evidence consumed by the audit workflow consists of public task artifacts, execution traces from our own evaluation runs which we released, public leaderboard trials, and issue reports that community members posted publicly to the Terminal-Bench repositories. We reference reporters through the public issues. Terminal-Bench 2.1 is released under the same Apache-2.0 license as Terminal-Bench 2.0, and every repair is documented with the supporting evidence so that maintainers and users can inspect or revert it. Because benchmark scores inform product claims we apply one rubric and one set of resource conventions to every task and agent, and we report per-agent score changes in full rather than only in aggregate.

The workflow uses LLM agents to propose defect labels. However no repair was merged without maintainer review, and we report agent agreement with the human-adjudicated repair set.

REPRODUCIBILITY STATEMENT

Terminal-Bench 2.1 is public. The audit workflow is described in Sec. 4, and Appx. A gives the continuous-validation checks. The evaluation setup is in Appx. B: the models (Appx. B.1), the harnesses (Appx. B.2), and the per-agent configurations (Appx. B.3). Per-task pass-rate changes between TB2.0 and TB2.1 are listed in Appx. C.1.

REFERENCES

Reem Aleithan, Haoran Xie, David Lo, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2410.06992.

Anthropic. Claude code: An agentic coding tool. Software documentation, 2025. URL https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/overview. CLI reference: https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/cli-reference.

Anthropic. Quantifying infrastructure noise in agentic coding evals. Anthropic Engineering Blog, 2026. URL https://www.anthropic.com/engineering/infrastructure-noise.

Anthropic. Introducing Claude Opus 4.6. Anthropic announcement and system card, February 5 2026a. URL https://www.anthropic.com/news/claude-opus-4-6. Terminal-Bench 2.0 reported on Terminus-2 harness (Codex CLI for OpenAI models), 1× guaranteed / 3× ceiling resource allocation, 5–15 samples per task across staggered batches.

Anthropic. Introducing Claude Sonnet 4.6. Anthropic announcement and system card, February 17 2026b. URL https://www.anthropic.com/news/claude-sonnet-4-6.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. Evaluating large language models trained on code. *arXiv preprint*, 2021. URL https://arxiv.org/abs/2107.03374.

Cursor Research Team. Composer 2: A fast frontier coding model built for the agent loop. Cursor technical report, 2026. URL https://cursor.com/resources/Composer2.pdf. Reports Terminal-Bench 2.0 with mixed harnesses (Claude Code for Anthropic models, Simple Codex for OpenAI, Harbor for Composer-2), averaged over 5 iterations.

Google. Gemini CLI: An open-source AI agent for the terminal. Open-source software (Apache 2.0), 2025. URL https://github.com/google-gemini/gemini-cli. Documentation: https://geminicli.com/docs/; tool reference: https://geminicli.com/docs/tools/.

Google DeepMind. Gemini 3 Flash: Model card. Technical report, Google DeepMind, December 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf.

Google DeepMind. Gemini 3.1 Pro: Model card. Technical report, Google DeepMind, February 2026a. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf.

Google DeepMind. Gemma 4: Open multimodal models. Model card and announcement; https://ai.google.dev/gemma/docs/core/model_card_4, 2026b. URL https://deepmind.google/models/gemma/gemma-4/. Released April 2026. Sizes: E2B, E4B, 26B MoE, 31B Dense; 256K context. We use the 31B Dense variant in the broader 20-pair audit pool.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, et al. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2403.07974.

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.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S. Siegel, Boyi Wei, Tianci Xue, Ziru Chen, et al. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2510.11977.

Kimi Team. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026. URL https://arxiv.org/abs/2602.02276.

LangChain. Langsmith: Ai agent and llm observability platform. https://www.langchain.com/langsmith/observability, 2026. Accessed: 2026-06-30.

Kevin Meng, Vincent Huang, Jacob Steinhardt, and Sarah Schwettmann. Introducing docent, 2025. URL https://transluce.org/introducing-docent. Transluce AI Technical Report.

Mike A. Merrill, Alexander G. Shaw, et al. Terminus: The terminal-bench baseline agent, 2025. URL https://github.com/terminal-bench/terminus. Baseline agent scaffold for Terminal-Bench.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

OpenAI. GPT-4 technical report, 2024a. URL https://arxiv.org/abs/2303.08774.

OpenAI. Introducing SWE-bench Verified. OpenAI blog post, 2024b. URL https://openai.com/index/introducing-swe-bench-verified/. Human-validated 500-task subset of Jimenez et al. (2024).

OpenAI. Codex CLI: A lightweight coding agent that runs in the terminal. Software platform, 2025a. URL https://openai.com/codex. CLI reference: https://developers.openai.com/codex/cli/reference/.

OpenAI. Ghosts in the codex machine. Technical report, 2025b. OpenAI technical report.

OpenAI. GPT-5.3-Codex system card. Technical report, OpenAI, 2026a. URL https://deploymentsafety.openai.com/gpt-5-3-codex/gpt-5-3-codex.pdf.

OpenAI. GPT-5.4 Thinking system card. Technical report, OpenAI, 2026b. URL https://deploymentsafety.openai.com/gpt-5-4-thinking/gpt-5-4-thinking.pdf.

OpenAI. GPT-5.5 system card. Technical report, OpenAI, 2026c. URL https://deploymentsafety.openai.com/gpt-5-5/gpt-5-5.pdf.

OpenAI. Harness engineering: leveraging Codex in an agent-first world. https://openai.com/index/harness-engineering/, February 2026d. Accessed: 2026-05-05.

Qwen Team. Qwen3.5-27B model card. Hugging Face model card, 2026a. URL https://huggingface.co/Qwen/Qwen3.5-27B/blob/fc05daec18b0/README.md. Revision fc05daec18b0a78c049392ed2e771dde82bdf654.

Qwen Team. Qwen3.5-9B model card. Hugging Face model card, 2026b. URL https://huggingface.co/Qwen/Qwen3.5-9B/blob/c20223623576/README.md. Revision c202236235762e1c871ad0ccb60c8ee5ba337b9a.

Qwen Team. Qwen3.6 model card. Hugging Face, 2026c. URL https://huggingface.co/Qwen/Qwen3.6-27B. Terminal-Bench 2.0 reported on Harbor/Terminus-2; 3h timeout, 32 CPU/48 GB RAM, max_tokens=80K, 256K context, averaged over 5 runs. Different harnesses for SWE-Bench (internal scaffold), SkillsBench (OpenCode), and NL2Repo (Claude Code).

Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? *Proceedings of ICML*, 2019. URL https://arxiv.org/abs/1902.10811.

ServiceNow Research. WebArena Verified: Reliable evaluation for web agents. Software release, 2026. URL https://github.com/ServiceNow/webarena-verified. Audited 812-task subset of Zhou et al. (2024).

Lin Shi, Haowei Lin, Zixuan Zhu, Xiaoyue Zhou, and Xiang Li et al. Harbor adapters and harbor-index: Infrastructure and a curated meta-dataset for large-scale agentic evaluation, 2026. URL https://arxiv.org/abs/2609.04298.

Terminal-Bench Team. Terminal-bench: GitHub issue and pull-request tracker, 2026a. URL https://github.com/harbor-framework/terminal-bench. Accessed May 2026; community-reported defects against the public TB and TB2.0 task sets.

Terminal-Bench Team. Terminal-bench 2.0: GitHub issue and pull-request tracker, 2026b. URL https://github.com/harbor-framework/terminal-bench-2. Accessed May 2026; community-reported defects against the public TB2.0 task set.

Xingyao Wang, Boxuan Chen, Ziniu Adelt, Yufan Shi, Bowen Liu, Shuyan Yang, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-free LLM benchmark. In *The Thirteenth International Conference on Learning Representations*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

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

Boxi Yu et al. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. URL https://arxiv.org/abs/2506.09289.

Z.ai. Terminal-bench 2.0 verified: Instruction & environment fix version. Hugging Face dataset, https://huggingface.co/datasets/zai-org/terminal-bench-2-verified, 2026. Documents 89 environment fixes and 11 instruction fixes; reports a 5.5 pp shift on Claude Opus 4.5 from environment fixes alone. Companion to GLM-5 (Zeng et al., 2026).

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint*, 2026. URL https://arxiv.org/abs/2602.15763.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, et al. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2307.13854.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, et al. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2507.02825.

APPENDIX CONTENTS

A Audit Workflow Details **15**

A.1 Continuous-validation pipeline 15

A.2 Audit Workflow Evaluation 15

B Terminal-Bench Benchmark Evaluation **16**

B.1 Models evaluated 16

B.2 Harness Evaluation 16

B.3 Agent configuration overrides 17

C Supplementary results **19**

C.1 Per-task pass-rate changes (TB2.0 → TB2.1) 19

C.2 Agent Failure Decomposition 19

C.3 Case study of 2 Harbor-Index tasks 20

Table 2: **Failure-modes among the 28 tasks revised in TB2.1.**

Instruction–verification mismatch (16)	**Environment issues (8)**	**Task drift (12)**
adaptive-rejection-sampler, build-pmars, caffe-cifar-10, configure-git-webserver, filter-js-from-html, fix-git, hf-model-inference, install-windows-3.11, mteb-leaderboard, mteb-retrieve, polyglot-c-py, polyglot-rust-c, query-optimize, sam-cell-seg, torch-tensor-parallelism, train-fasttext	caffe-cifar-10, compile-compcert, crack-7z-hash, filter-js-from-html, gpt2-codegolf, query-optimize, torch-pipeline-parallelism, torch-tensor-parallelism	adaptive-rejection-sampler, build-pmars, build-pov-ray, extract-moves-from-video, financial-document-processor, make-doom-for-mips, mcmc-sampling-stan, mteb-leaderboard, mteb-retrieve, overfull-hbox, protein-assembly, rstan-to-pystan

A Audit Workflow Details

A.1 Continuous-validation pipeline

As benchmark users continue to open issues and pull requests, maintainers need a lightweight way to check whether proposed changes preserve task validity. We implement this audit workflow with GitHub Actions that can be invoked manually or triggered automatically on pull requests for a given task.

The pipeline exposes two checks:

- `/audit` calls the auditor agent to review the task instructions and available trial logs.
- `/oracle` builds the task environment and runs the reference solution, checking that a known-correct solution still passes.

Propose-only. Although agents could be given permission to propose fixes and merge pull requests automatically, we deliberately restrict the workflow to propose-only operation. Each run is capped at five repair iterations, and the workflow grants pull-requests but prevents agents from directly changing repository state. Instead, proposed fixes are emitted as maintainer-reviewable diffs in `audit/changes/<task>.diff`.

The audit rubrics are integrated in https://github.com/harbor-framework/benchmark-template.

A.2 Audit Workflow Evaluation

We measure agreement at two levels. Managed coding assistants, Claude Code (Opus 4.6) and Codex (GPT-5.4), agree on the primary defect category, including *no defect*, at $\kappa = 0.80$ over all 89 tasks. Lower-cost API auditors are evaluated on the binary triage decision (flagged vs. clean) over all 89 TB2.0 tasks (Table 1), with pairwise agreement of $\kappa = 0.55$–0.77. Per category (one-vs-rest), their agreement is moderate for instruction–verification mismatch ($\kappa = 0.60$) and environment issues ($\kappa = 0.59$) and low for task drift ($\kappa = 0.21$–0.41). Because every drift defect breaks the reference solution, the deterministic `/oracle` check catches drift regardless of auditor labels.

Table 1: **Inter-judge agreement.** Cohen's κ between pairs of judges on the binary flag (flagged vs. audit-clean) over all 89 TB2.0 tasks.

Agent A	Agent B	Agree	Disagree	κ
gpt-4o-mini	gpt-4o	69	20	0.550
gpt-4o-mini	gemini-2.5-flash	79	10	0.769
gpt-4o	gemini-2.5-flash	69	20	0.549

Per-task map to error categories Table 2 shows the primary failure mode assigned to each task found to be broken. Table 3 shows the final 28 revised tasks, and which component in the task was updated for TB 2.1.

Table 3: **Breakdown of the 28 TB 2.0 tasks revised in TB 2.1, by the stage of the task definition that required repair.**

Task	Instr.	Env.	Oracle	Tests	Time	Mem.	CPUs
adaptive-rejection-sampler	✓		✓				
build-pmars	✓			✓			
build-pov-ray			✓				
caffe-cifar-10	✓	✓		✓	✓	✓	✓
crack-7z-hash					✓	✓	
compile-compcert		✓					✓
configure-git-webserver			✓	✓			
extract-moves-from-video	✓	✓					
filter-js-from-html	✓				✓	✓	
financial-document-processor			✓				
fix-git		✓		✓			
gpt2-codegolf						✓	
hf-model-inference				✓			
install-windows-3.11	✓			✓			
make-doom-for-mips			✓				
mcmc-sampling-stan				✓			
mteb-leaderboard	✓	✓				✓	
mteb-retrieve	✓	✓					
overfull-hbox			✓				
polyglot-c-py				✓			
polyglot-rust-c				✓			
protein-assembly		✓	✓				
query-optimize	✓				✓		
rstan-to-pystan				✓			
sam-cell-seg	✓						
torch-pipeline-parallelism						✓	
torch-tensor-parallelism	✓					✓	
train-fasttext				✓			

B TERMINAL-BENCH BENCHMARK EVALUATION

B.1 MODELS EVALUATED

The evidence data included leaderboard trials from the 13 models listed in Table 4.

Table 4: **Models evaluated.** One row per model entry used in the agent-rank comparison or compute-scaling analyses. CTX values in italics are pinned smaller by the harness config; plain values are the vendor maximum.

Model	Provider	Route	Ctx
Claude Opus 4.6 (Anthropic, 2026a)	Anthropic	Anthropic API	200K
Claude Sonnet 4.6 (Anthropic, 2026b)	Anthropic	Anthropic API	200K
GPT 5.3 Codex (OpenAI, 2026a)	OpenAI	OpenAI API	400K
GPT 5.4 (OpenAI, 2026b)	OpenAI	OpenAI API	400K
GPT 5.4 mini (OpenAI, 2026b)	OpenAI	OpenAI API	400K
GPT 5.5 (OpenAI, 2026c)	OpenAI	OpenAI API	400K
Gemini Flash 3 (Google DeepMind, 2025)	Google	Google API	1M
Gemini Pro 3.1 (Google DeepMind, 2026a)	Google	Google API	1M
Gemma 4 31B (Google DeepMind, 2026b)	Google	Together.ai	256K *(131K)*
GLM 5.1 (Zeng et al., 2026)	Z.AI	Anthropic-shape	200K
Kimi K2.6 (Kimi Team, 2026)	Moonshot AI	Together.ai	256K *(131K)*
Qwen 3.5 9B (Qwen Team, 2026b)	Alibaba	Together.ai	254K *(131K)*
Qwen 3.5 27B (Qwen Team, 2026a)	Alibaba	Together.ai	254K

B.2 HARNESS EVALUATION

Models were evaluated with a common baseline harness, `terminus-2` (Merrill et al., 2026), and, where available, the model provider's preferred agent harness. The latter includes Anthropic's Claude Code CLI (Anthropic, 2025), OpenAI's Codex CLI (OpenAI, 2025a; 2026d), and Google's Gemini CLI (Google, 2025).

Table 5: **Harnesses evaluated.**

Harness	Scaffold	Source / defaults
`terminus-2`	single-agent ReAct (Yao et al., 2023)	Terminal-Bench 2.0 baseline (Merrill et al., 2025; Terminal-Bench Team, 2026a); including JSON-action parser, shell + file I/O tools. **Defaults:** `enable_summarize=True`, `proactive_summarization_threshold=8000`, `temperature=1.0` for thinking-capable models.
`claude-code`	provider-managed agentic CLI	Anthropic Claude Code, headless (Anthropic, 2025); self-managed tool palette + context compaction. **Defaults in our runs:** `reasoning_effort=high`, adaptive thinking on.
`codex`	provider-managed agentic CLI	OpenAI Codex CLI, non-interactive (OpenAI, 2025a; 2026d); native Responses-API tool calls, managed context. `use_responses_api=True`.
`gemini-cli`	provider-managed agentic CLI	Google Gemini CLI, Apache 2.0 (Google, 2025); shell + file I/O tools, managed context.

B.3 Agent configuration overrides

Table 6 lists the agent configurations. Rows marked "default" use the harness-level defaults documented in Table 5.

We note that `terminus-2` triggers proactive summarization when the free context drops below a threshold (default 8K tokens). We leave the default for frontier-API models and GLM 5.1, where a single turn rarely exceeds 8K tokens. For the open-weight long-context models (Gemma 4 31B and Kimi K2.6, both with 256K context windows), we raise the threshold to 131K tokens after observing trials where summarization fired too late to leave room for one more turn.

The Gemma 4 31B and Kimi K2.6 rows in Table 6 are collapsed for readability; the full sampling settings are:

- **Gemma 4 31B:** temperature 1.0, interleaved thinking on, max input tokens 253,952, max output tokens 8,192, proactive-summarization threshold 131,072, `top_p`= 0.95, `top_k`= 20, `presence_penalty`= 1.5.
- **Kimi K2.6:** temperature 0.6, interleaved thinking on, max input tokens 253,952, max output tokens 8,192, proactive-summarization threshold 131,072, `top_p`= 0.95, `top_k`= 20.

Table 6: **Agent–model configurations.** "Default" means the harness-level defaults of Table 5 apply unchanged. Additional overrides are listed below.

Agent	Model	Overrides vs. harness default
claude-code	Claude Opus 4.6	`reasoning_effort=high`
claude-code	Claude Sonnet 4.6	`reasoning_effort=high`
claude-code	GLM 5.1	default
codex	GPT 5.3 Codex	`reasoning_effort=xhigh`
codex	GPT 5.4	`reasoning_effort=xhigh`
codex	GPT 5.4 mini	`reasoning_effort=xhigh`
codex	GPT 5.5	`reasoning_effort=xhigh`, `temperature=1.0`
gemini-cli	Gemini Flash 3	default
gemini-cli	Gemini Pro 3.1	default
terminus-2	Claude Opus 4.6	`reasoning_effort=high`, `T=1`
terminus-2	Claude Sonnet 4.6	`reasoning_effort=high`, `T=1`
terminus-2	GLM 5.1	`T=1.0`, `thinking.type=enabled`
terminus-2	GPT 5.3 Codex	`reasoning_effort=xhigh`, `use_responses_api=True`
terminus-2	GPT 5.4	`reasoning_effort=xhigh`, `use_responses_api=True`
terminus-2	GPT 5.4 mini	`reasoning_effort=xhigh`, `use_responses_api=True`
terminus-2	GPT 5.5	`reasoning_effort=xhigh`, `T=1`, `use_responses_api=True`
terminus-2	Gemini Flash 3	`reasoning_effort=high`
terminus-2	Gemini Pro 3.1	`reasoning_effort=high`
terminus-2	Gemma 4 31B	`T=1.0`, `interleaved_thinking=True`; open-weight pin block (see text)
terminus-2	Kimi K2.6	`T=0.6`, `interleaved_thinking=True`; open-weight pin block (see text)

C SUPPLEMENTARY RESULTS

C.1 PER-TASK PASS-RATE CHANGES (TB2.0 → TB2.1)

Table 7 reports the per-task pass-rate change from TB2.0 to TB2.1, averaged across all agents. After the changes no task remains unsolved. The largest regression is in task `overfull-hbox`: in TB2.0 a broken environment build left the verifier without ground truth, crediting outputs that the repaired verifier rejects.

Table 7: **Changed-task pass rates.** Rates aggregated across all agent model pairs evaluated.

Task	TB2.0	TB2.1
adaptive-rejection-sampler	31.7%	58.3%
build-pmars	75.0%	81.7%
build-pov-ray	75.0%	68.3%
caffe-cifar-10	5.0%	69.2%
compile-compcert	45.0%	48.3%
configure-git-webserver	46.7%	50.0%
crack-7z-hash	75.0%	86.7%
extract-moves-from-video	8.3%	13.3%
filter-js-from-html	0.0%	6.7%
financial-document-processor	61.7%	56.7%
fix-git	98.3%	95.0%
gpt2-codegolf	15.0%	18.3%
hf-model-inference	88.3%	86.7%
install-windows-3.11	0.0%	23.3%
make-doom-for-mips	6.7%	1.7%
mcmc-sampling-stan	75.0%	71.7%
mteb-leaderboard	28.3%	36.7%
mteb-retrieve	15.0%	41.7%
overfull-hbox	53.3%	36.7%
polyglot-c-py	5.0%	80.0%
polyglot-rust-c	5.0%	66.7%
protein-assembly	20.0%	15.0%
query-optimize	50.0%	43.3%
rstan-to-pystan	58.3%	58.3%
sam-cell-seg	0.0%	53.3%
torch-pipeline-parallelism	8.3%	10.0%
torch-tensor-parallelism	15.0%	66.7%
train-fasttext	5.0%	3.3%

C.2 AGENT FAILURE DECOMPOSITION

Agentic evaluation infrastructure is still a work in progress: runner failures, provider interruptions, authentication errors, sandbox crashes, and setup timeouts can all occur before the agent has produced a task-level outcome. To avoid mixing these infrastructure failures with model behavior, this analysis uses a conservative whitelist-filtering approach. The table builder retains only trials whose terminal state can be interpreted as an agent outcome: a pass, a completed wrong answer, or an `AgentTimeoutError`. Other upstream infrastructure errors are filtered before the decomposition is computed. However, we note that these filtered infrastructure trials account for roughly 27% of upstream trials.

Fig. 7 decomposes each whitelisted trial into one of three mutually exclusive outcomes: a pass (`reward` $= 1$), a completed but incorrect answer (`reward` $= 0$), or an agent timeout (`AgentTimeoutError`). At fixed budget, the weakest agents are dominated by timeouts rather than by scored wrong answers. This timeout-dominant pattern appears for `terminus-2` paired with Qwen 3.5 9B, Gemma 4 31B, Kimi K2.6, and GPT 5.4 mini. For these cells, the benchmark often censors the run before a final answer can be scored. These results motivate the budget sweeps in Sec. 6.2.

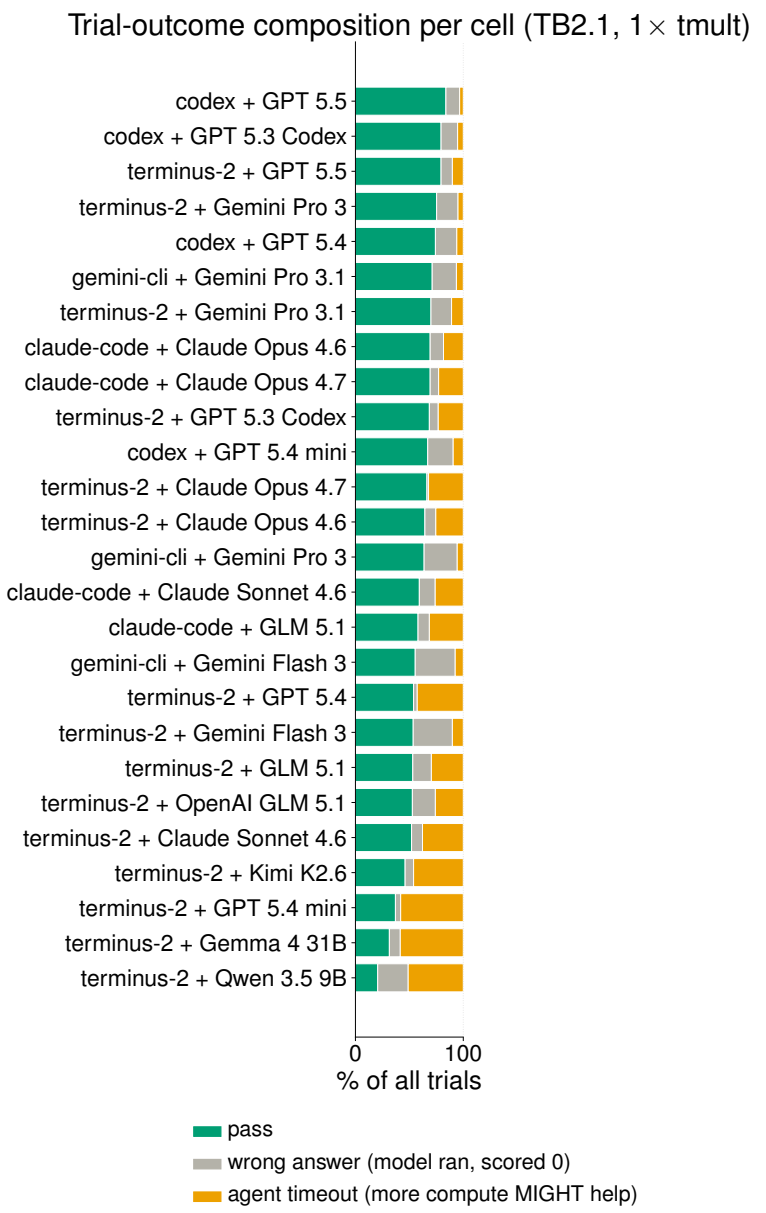

Figure 7: **Trial outcomes by agent.** Each horizontal bar shows, for one agent, the fraction of trials that passed (green), returned a clean wrong answer (gray), or hit `AgentTimeoutError` (amber).

C.3 Case study of 2 Harbor-Index tasks

C.3.1 A GAIA2 task

The tasks and how they are graded. GAIA2 benchmark evaluates an agent inside a simulated multi-application environment (Email, Calendar, Contacts, Chats, Shopping, and others) originally provided by Meta's Agents Research Environments (ARE). Upstream, ARE runs its own agent loop and grades a trajectory with a single event-graph judge that jointly checks the oracle's write-action graph, at exact per-tool call counts, together with the agent's final user-facing message. To run these tasks under Harbor framework, we re-implement the ARE environment as a sidecar bridge: each ARE app method is registered as an individual MCP tool, and a background loop advances simulated time so that scripted environment reactions, such as a counterpart replying to an email the agent has just sent, fire in response to the agent's actions. This GAIA2 adapter, however, does not fully replicate ARE's original environment and judge, and the resulting fidelity gaps are the source of the

bugs below. As a concrete example, `gaia2-ambiguous` issues one instruction that bundles four requests (remove every saved apartment listed below $1,500, buy a board game, reply to a contact's latest email, and schedule a weekend grocery trip). The intended behavior is to carry out the three determinate requests as write actions and to surface the ambiguous one as a clarifying question rather than guess.

Why it fails. The continuous-validation workflow reported 3 bugs with the task that can fail a correct agent or reward an incorrect one. (1) *Byte-exact reaction matching.* To decide whether to fire a dependent environment reaction, the sidecar matched the agent's tool call against the oracle event by byte-exact argument equality. For actions whose arguments are free-form text that the agent must phrase itself, such as an email subject or body or a chat message, a correct but differently worded call never matched, so the scripted reaction never fired and the agent was starved of the follow-up events it needed to finish, scoring 0 through no fault of its own. (2) *Answer-only judging.* The ambiguity capability was routed to a grader that inspected only the agent's final message and ignored the requested write actions, so an agent that skipped every deterministic task but asked the right clarifying question still scored a perfect 1.0. (3) *Failed oracle solution.* The oracle keyed follow-up actions, such as forwarding a specific email, to fixed identifiers, but the real entity IDs are minted freshly on every run, so genuine work compared against the stale placeholder could never match.

How we fix it. We patched this task on all three fronts. (1) We added an LLM-equivalence check on the free-form text responses. We also made identifier resolution dynamic: the sidecar records a provenance map from each event to the entity ID at runtime, and the verifier resolves the oracle's placeholders to those concrete IDs before comparison, rejecting any that remain unresolved. (2) We moved the ambiguity capability from the answer-mode set to the action-mode set, so it is now scored by a list of write actions instead of the last action only. (3) The reference oracle is patched from a single action to all seven write actions the task demands, so that it faithfully verifies the correctness of agent actions.

C.3.2 An OmniMath task

The task and how it is graded. `omnimath-find-perfect-square-functions` is an OmniMath competition problem. The agent is given one functional-equation prompt, to determine all functions $f\colon \mathbb{Z} \to \mathbb{Z}$ such that $f(f(a) - b) + b\,f(2a)$ is a perfect square for all integers a and b, and must write a complete characterization of the solution set as free-form mathematical equation to a response file. Because no closed-form checker can decide whether two natural-language characterizations are equivalent, the task is graded by an LLM verifier that compares the agent's answer against a gold answer. For this problem, the gold answer has two distinct solution families: $f(n) = n^2$ for every integer n, and the parity-dependent family with $f(n) = 0$ for even n and $f(n)$ any perfect square for odd n. A correct answer must identify both families.

Why it fails. As flagged by the continuous-validation workflow, the main defect is that the judge prompt never stated what a correct answer had to contain. It supplied the question, the target answer, and the agent's answer and asked the model for a binary score, with no definition of correctness and no requirement that an answer to a "determine all" question cover every solution family. For this problem, the verifier was therefore free to accept an answer that gave only one of them, and its judgments on such partial answers were not stable. The grading path was also brittle for reasons unrelated to the mathematics: the stored ideal answer held LaTeX whose backslashes were not escaped for JSON, so a control sequence such as `\text` was read as a literal tab; an empty judge base URL variable could crash the API client before grading began; and the reward file did not always match the scalar schema Harbor reads.

How we fix it. We rewrite the grading prompt to state the expected output explicitly. The judge is now instructed to mark an answer correct only when it is mathematically equivalent to the target and complete, to require that every distinct solution, family, or case in the target be present, to treat a partial answer to a "determine all" or "find all" question as incorrect even when the solutions it does give are valid, and to disregard differences of notation, ordering, and wording; it also returns its reasoning together with the binary score. We also hardened the JSON parsing logic in the grader so the intended LaTeX formula could survive, removed empty base URL variables that crashed the client, and standardized the verifier on the scalar reward file Harbor reads.

Benchmark	Task
HLE (10)	hle-artist-augustine-quote hle-concentric-circles-dataset hle-dirac-fermion-tunneling hle-fibered-category-schemes hle-identify-city-from-photo hle-identify-ingvar-runestone hle-interval-coverage-bound hle-name-alkaloid-compound hle-shock-wave-density-profile hle-vowel-marking-system
GSO (7)	gso-speedup-hf-datasets gso-speedup-numpy-strings gso-speedup-pandas-merge gso-speedup-pandas-period-fmt gso-speedup-pandas-seq-to-range gso-speedup-pillow-filter gso-speedup-pydantic-enum
GAIA2 (6)	gaia2-adapt-hard-1 gaia2-adapt-hard-2 gaia2-adapt-med gaia2-ambiguous gaia2-timed-1 gaia2-timed-2
LabBench (5)	labbench-count-deg-in-pathway labbench-habenula-fluorescence-change labbench-highest-auc-neuron-set labbench-read-asap2f-step-response labbench-survival-curve-at-25d
QCircuitBench (3)	qcircuitbench-design-qaoa-maxcut-n8 qcircuitbench-design-ternary-simon-circuit-n3 qcircuitbench-design-ternary-simon-oracle-n2
OmniMath (2)	omnimath-find-perfect-square-functions omnimath-maximize-gf-sum
GPQA Diamond (1)	gpqadiamond-cope-rearrangement-products

Table 8: The 34 Harbor-Index tasks audited with the continuous-validation workflow, grouped by source benchmark.