Auditing Terminal-Bench Shows Stable Rankings but Shifted Efficiency

Anonymous Authors [1]

Abstract

Coding agents are among the most economically consequential deployments of LLMs to date. The benchmarks we use to evaluate agents are load-bearing, supporting rapid improvement in model capabilities. However, agentic benchmarks accumulate defects on every axis: instructions can be overspecified, verifiers can be underspecified, and environments drift as external dependencies change after release. We audit Terminal-Bench 2.0 with a continuous-validation pipeline (automated CI checks, LLM-as-judge). We find task-side defects in 28 of 89 tasks (31%) in the benchmark. We ship the fixes as Terminal-Bench 2.1. Across 19 agents, scores rise by up to 12 pp, but agent ranking is stable (Spearman $\rho = 0.958$). On the changed tasks, 28.8% of the tokens TB 2.0 charged to failed trials on changed tasks were spent on what TB 2.1 verifies as solvable work. Cost-aware curves show that the audit shifts per-agent efficiency in both directions: some agents gain coverage while becoming more token-efficient, others while becoming less. On TB 2.1 benchmark, switching from `codex` to `terminus-2` at fixed OpenAI model drops accuracy by up to 29.9 pp; raising the per-task budget closes the gap, indicating the harness effect is wall-clock-bound rather than capability-bound

1. Introduction

Agentic benchmarks decay in ways static benchmarks do not: external APIs change under the task, environment variance leaks into resource budgets, and agents find solutions task authors did not anticipate. The defect rate is high enough that even carefully curated benchmarks accumulate substantial maintenance debt: Terminal Bench 2.0 (Merrill et al., 2026) has over 50 open Github issues (as of May 2026) tagged as task defects (Laude Institute, 2026a;b; Zeng et al., 2026).

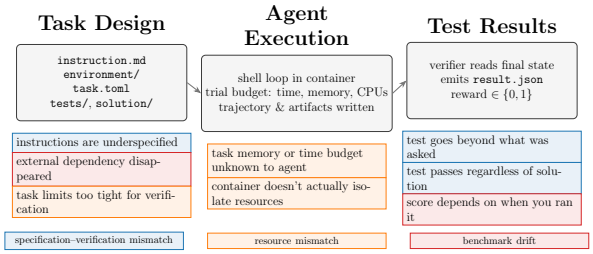

Figure 1. Every stage of an agentic-benchmark task is a place where things can fail; colors mark the three failure categories (blue = specification–verification mismatch, orange = resource, red = drift).

Without a methodology for verifying that proposed patches cover the full defect set, reported model improvements on agentic benchmarks cannot be cleanly separated from underlying capability gains.

Leaderboard scores are increasingly load-bearing for cross-vendor model comparisons, and Terminal Bench 2.0 is cited in technical reports from multiple providers including OpenAI (OpenAI, 2024a), Anthropic (Anthropic, 2026a), Cursor (Cursor Research Team, 2026), Qwen (Qwen Team, 2026), GLM (Zeng et al., 2026), Google (Google DeepMind, 2026) among others. However, technical reports have began to report Terminal Bench 2.0 scores under different resource constraints (memory, time, cpu count), so it is unclear how to compare the headline numbers in these reports to the original leaderboard.

Auditing agentic benchmarks poses a harder validation problem than evaluating a static benchmark. Each task spans multiple components, an instruction, a containerized environment, a verifier, and a resource budget; and a defect in any component could produce a misleading pass rate (Fig. 1). Task-level defects are entangled with serving infrastructure and harness configuration: Segato (2026) reports 6pp swings from using different virtual machine providers and OpenAI (2025b) documents performance variance from hardware heterogeneity, and timeout-retry behavior within a single vendor's deployment.

In this work, we describe the process to audit Terminal Bench 2.0. Three prior approaches to benchmark validity inform ours. Recht et al. (2019) validate static benchmarks by re-sampling and measuring rank correlation between the

[1]Affiliation. Correspondence to: Anonymous Authors <email>.

Proceedings of the 43^{rd} International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026.

original benchmark and a resampled benchmark; we adopt the rank-correlation check as a test of construct preservation under repair rather than under re-sampling. Verified-style relabeling (OpenAI, 2024b) produces validated benchmark subsets, but the one-shot "Verified" framework assumes a terminal validated state that does not apply to benchmarks tied to live infrastructure. (Jain et al., 2025) proposes to continuously monitor and update problems. However, high quality tasks are already nontrivial to collect. Recent work repairs 11 Terminal Bench 2.0 tasks alongside the GLM-5 release(Zeng et al., 2026). However, it remains unclear if their approach fixes only a subset of the problems in the benchmark. Coding agents change the economics of audits: agent-driven triage indexes task instructions, tests, traces, and community reports at low marginal cost, with internet and memory restrictions preventing solution leakage.

We build an auditing pipeline on GitHub workflows and apply it to Terminal Bench 2.0. The pipeline allows us to combine community reports (github and the web) with an agent-driven audit grounded in public-leaderboard trials from 21 third-party agents (Sec. 2). Looking at community reports and diverse model traces allows us to better capture both false positive and false negative trials. Having collected evidence from these sources, agents summarize, rank and propose repairs to specific tasks, which a human maintainer can review before merging (Sec. 3.1). Across the 89 tasks in TB2.0, the pipeline surfaces 28 (31%) with at least one task-side defect, classified into three categories: *specification–verification mismatches* (16 tasks), where the verifier silently encoded constraints the instruction did not state; *resource mismatches* (8 tasks), where per-task budgets fit the reference solution but starved alternative strategies; and *benchmark drift* (12 tasks), where external dependencies changed after release (Sec. 3.2). We release the patched benchmark as Terminal Bench 2.1.

We then re-evaluate 19 agents in both TB 2.0 and TB 2.1 and find that agent ranks remain highly correlated, with bounded but nonzero movement (Sec. 4.1). The largest per-agent gains come from the audit removing specification–verification mismatches. We note that resource budgets are themselves a legitimate dimension of agent evaluation, they measure efficiency, which deployment cares about as much as success. The audit does not relax this pressure; rather, it enforces a feasibility floor, ensuring that the per-task budget leaves headroom for the verifier to run the oracle solution. Budgets below the verification threshold do not measure efficiency, because every agent fails regardless of capability. TB2.1's resource fixes raise such budgets to feasibility while preserving the efficiency signal everywhere else.

Where are the remaining failures in Terminal Bench 2.1? Even with feasibility restored, we then observe timeout errors can account for up to 50% of failures for some agents. Agents are not given information on how long they have to complete a task in Terminal Bench 2.0. While some work has found that sharing this information can be useful(Liu et al., 2025), we leave the disclosure question to future work and instead ask a simpler counterfactual: how does agent performance change as we vary the per-task time budget? This isolates the effect of the budget itself from the effect of agent budget-awareness.

Cost-aware coverage curves (Sec. 4.2) show different time and token efficiency frontiers in small versus large models (as expected) and between TB2.0 and TB2.1. Terminal Bench 2.1 leads to improved efficiency frontiers for larger agents, and unchanged for small agents. For any agent it can change wall-clock (a sensitive metric) but not on total tokens. The same confound appears at the harness level. Within-vendor harness swaps (Sec. 4.3) move per-agent scores by margins comparable to a model upgrade: switching from `codex` to `terminus-2` on OpenAI models drops accuracy by 10.9–29.9 pp. Failure-mode decomposition shows this gap is wall-clock-bound (terminus-2 produces 6–18 timeout-bound tasks per model versus 0–2 under codex), and raising the per-task budget closes most of it. The codex-to-terminus-2 gap on OpenAI models, previously read as evidence of scaffold-design advantage, is primarily a budget effect in TB2.1.

2. Preliminaries

2.1. Related Work

Validity audits of evaluation benchmarks. Agentic benchmarks accumulate defects and the field has converged on three responses. The first is re-collection: Recht et al. (2019) re-drew ImageNet and CIFAR test sets and used rank correlation between the old and new draws as evidence the new test measured the same construct, the methodology we adopt here. The second is audited-and-relabeled subsets: SWE-Bench (Jimenez et al., 2024) problems were filtered in Verified (OpenAI, 2024b), with contamination audits in Aleithan et al. (2024) and Yu et al. (2025); WebArena (Zhou et al., 2024) likewise has Verified (ServiceNow Research, 2026); GAIA (Mialon et al., 2023), MLE-Bench (Chan et al., 2025), τ-Bench (Yao et al., 2024), and recent benchmark-noise quantification work (Plesner et al., 2026; Zhu & Kang, 2026) extend this line. The third is time-windowing: LiveCodeBench (Jain et al., 2025) date-stamps problems and evaluates on contamination-free slices. Post-hoc trace inspection tools such as Docent (Meng et al., 2025) make these audits tractable. Zhu et al. (2025)'s Agentic Benchmark Checklist (ABC) codifies the validity properties audits can target. OpenAI (2025b) documents the dual failure mode on the agent side: silent reward hacks and underspecified-task exploits that pass tests without solving the underlying problem. Our spec–verification mismatch category targets the same failure mode from the benchmark side.

LLM-as-judge as an audit primitive. LLM-as-judge has become a standard substitute for human labeling in evaluation pipelines (Zheng et al., 2023; Liu et al., 2023), with subsequent work examining how to align judges with human preferences and validate them at scale (Shankar et al., 2024; Thakur et al., 2024) and documenting systematic failure modes such as preference leakage (Cohen et al., 2025). Our audit pipeline uses LLM-as-judge as one signal alongside automated CI checks, calibrated against human review (Sec. 3) rather than treated as a final arbiter.

Agent harness as an evaluation axis. ReAct (Yao et al., 2023) cast scaffold design as a prompt-engineering target; AgentBench (Liu et al., 2024) documented that scaffold choice can move benchmark scores by tens of percentage points at fixed model. A family of open scaffolds has since emerged, including SWE-agent (Yang et al., 2024) and its lightweight variant mini-SWE-agent (SWE-agent Team, 2025), OpenHands (Wang et al., 2025), and the Terminus baseline shipped with Terminal-Bench (Merrill et al., 2025), alongside vendor-specific harnesses including Claude Code (Anthropic, 2025), OpenAI's Codex CLI (OpenAI, 2025a), and the Gemini CLI (Google, 2025). The original Terminal-Bench paper (Merrill et al., 2026) averages across model families and concludes scaffold is less impactful than model choice on average, but this can obscure within-family harness effects.

2.2. Problem setup

A Terminal-Bench task is a timed exam. An agent is given a problem description, the resources it needs to attempt the problem inside a sandboxed Linux container, and a grading script that checks its answer after the agent's time is up. Five components define the task:

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. If the oracle fails, the benchmark is broken, not the agent.

Test the grading code that runs after the agent stops.

Resources CPU, memory, and wall-clock budgets.

For each task, the five components are specified by the task contributors (Merrill et al., 2026). A defect in any of these components produces a misleading pass rate. Our audit (Sec. 3) classifies each TB2.0 defect by which component caused it.

Metrics and notation. An *agent* is a harness–model pairing and is the evaluation unit throughout the paper. We use "harness" for the scaffold component (the tool-use loop, prompt template, and orchestration code) and "model" for the underlying LLM weights. A *trial* is one independent attempt at a task by an agent. To evaluate efficiency, we use *timeout multipliers* to scale every task's wall-clock budget by a constant factor, the knob in Sec. 4.2.

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

Datasets. We operate with two datasets, one to be used in the audit pipeline and one as we evaluate the benchmarks. *Dataset 1* is external evidence on TB2.0: public leaderboard submissions, GitHub issues, prior independent audits, and other community-generated trial data collected by the agent from the web. The audit pipeline (Sec. 3) operates on Dataset 1 to surface candidate defects in TB2.0. *Dataset 2* is our own evaluation in TB2.0 and TB2.1: evaluation trials across 19 agents in the version-matched cohort. The results in Sec. 4 draw from that cohort (TB2.0 and TB2.1) and Scaling-4 subsets of it, where we increased the resource allocated for only a subset of models due to cost.

3. Auditing Terminal Bench 2.0

Sec. 3.1 describes the audit pipeline; Sec. 3.2 describes what it found. Because this pipeline can run continuously, TB2.1 is a checkpoint rather than a terminal state. We calibrated the pipeline on 15 community-flagged tasks and report performance against the 28 tasks that were ultimately revised in TB2.1. A detailed breakdown is in Appx. A.

Two prior efforts inform the design. SWE-bench Verified (OpenAI, 2024b) contracted professional developers to manually validate tasks along three axes: task specification, the tests, and the difficulty. Zhu et al. (2025)'s Agentic Benchmark Checklist (ABC) codifies validity guidelines for agentic evaluations. We mechanize both into a continuously-running pipeline with two design choices that differ from prior validation efforts. First, the pipeline repairs tasks rather than discards them, preserving benchmark coverage. Second, we version the result (TB2.0 $\rightarrow$ 2.1) rather than releasing a frozen "verified" snapshot, as benchmarks tied to live infrastructure cannot reach a terminal validated state.

3.1. Pipeline

The audit pipeline is illustrated in Fig. 2 and has four components: three evidence streams (**Detection**), a reconcile stage (**Reconciliation**), a draft-PR step (**Patch**), and a maintainer gate. Every flagged defect is reviewed by a

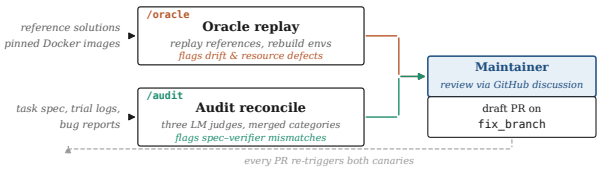

Figure 2. **TB-Audit on every PR.** Two GitHub Actions run in parallel: `/oracle` (programmatic) and `/audit` (LM-judges propose and categorize issues into the defect classes of Sec. 3.2). A maintainer can review the joint PR by observing the GitHub discussion thread.

human before merging for quality.

Detection. Our goal is to detect the category of the task defect and where it occurs. Different defect categories surface in different evidence. Specification–verification mismatches live in the verifier source; resource starvation appears in trial logs across many models. A *static reader* examines each task's specification (`task.toml`, instruction, Dockerfile, resource budgets). A *trace reader* examines trial logs from previous agent runs from the public leaderboard. A *web reader* searches GitHub PRs and the open web for community bug reports referencing the task. Each reader produces a checklist-based report grounded in the ABC validity guidelines, with citations for every flagged item.

Reconciliation. A reconcile prompt joins the three reports into a single ranked list of candidate defective tasks. Each candidate task carries a category label (one of the three classes defined in Sec. 3.2, or "no defect found") and a free-text summary of the suspected mechanism. The maintainer reviews this list rather than the underlying reports. Tasks can have multiple defects but we ask the LLM judge to report the primary defect.

Patch. Flagged candidates land as as draft PRs on a fix branch. The workflow has `pull-requests: write` but not `contents: write`; nothing merges without a maintainer.

Validation. The pipeline was validated using ground truth labels from 15 problems flagged by the community. Against the TB2.1 release labels, individual judge baselines reach 44–49% precision and 79–89% recall depending on judge model. Across three judges from two providers (`gpt-4o-mini`, `gpt-4o`, `gemini-2.5-flash`), cross-provider Cohen's κ on the binary flag reaches **0.77**, measuring flag stability rather than correctness. Requiring two judges to agree raises precision from 45.1% to 57.1% and lowers FPR from 45.9% to 24.6%, while recall falls from 82.1% to 71.4% (Table 2); consensus helps recover the release-diff labels but does not remove the need for review. Per-category κ reaches 0.60 on

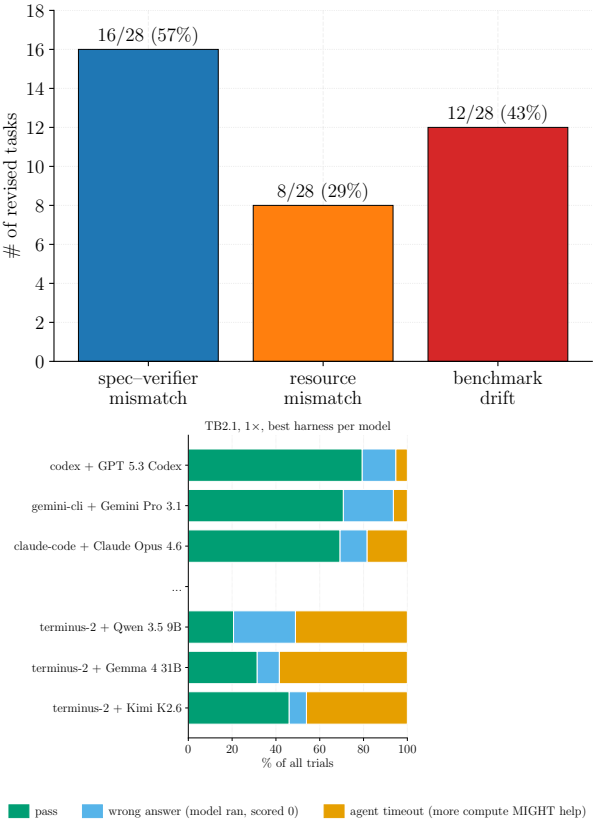

Figure 3. **Two complementary views of where TB2.1 failures come from.** (a) Task-side root-cause distribution among the 28 revised tasks (Table 4 for the per-task assignment). (b) Trial-outcome composition per agent cell on TB2.1 at 1× tmult. The rank correlation between low pass rate and high timeout share is the load-bearing observation: `terminus-2` cells with small models (Gemma, Qwen, GPT 5.4 mini, Kimi K2.6) sit at the bottom and are dominated by amber (compute-bound) failures, while `codex` and `gemini-cli` cells at the top fail mostly with wrong answers.

specification–verification mismatch and 0.59 on resource mismatch, but is low for drift detection. The complementary `/oracle` workflow catches drift deterministically: every drift defect modifies the environment or solution code, so re-running the oracle solution surfaces the defect.

Continuous operation. The pipeline operates as a github workflow and can run anytime. Existing tasks are re-audited when their environments drift; flagged defects land as draft PRs on a fix branch for maintainer and community review (Fig. 2). This mechanism differs from a one-shot curation pass like SWE-bench Verified, and supports a software-style versioning convention where each release is a numbered checkpoint rather than a re-branded snapshot.

3.2. Task level problems

The pipeline surfaced 28 of 89 tasks (31%) with task-side defects across three categories (Fig. 3). Eight tasks had

defects in more than one category; per-task labels are in Table 4. We report per agent accuracies in Appx. C.3.

Specification–verification mismatches (16 tasks, the most common). Either the instruction is underspecified, leaving out constraints that the test silently enforces, or the test is overspecified, encoding assumptions the task creator never realized they were making. In `query-optimize`, the instruction said not to modify the database, but the test checked a hash, an agent that that touched and reverted the file failed even though the database was logically identical Aleithan et al. (2024); Yu et al. (2025).

Benchmark drift (12 tasks). Tasks in TB2.0 use pre-built Docker images for reproducibility. However, tasks with internet access introduce external dependencies that change over time. We observe that evaluating the `/oracle` solution allows us to catch most of these drift. We note that Jain et al. (2025) treat temporal drift motivated by training-data contamination. We identified 12 tasks where external dependencies changed after the benchmark was built. We note that downstream effects often cascade, as a dependency change can shift what a correct solution looks like, how many steps it requires, or how much time and compute it takes to reach, in turn we observe that tasks not only get easier but also harder.

Resource mismatches (8 tasks). Setting a per-task budget requires predicting how an agent will behave. In TB2.0 task contributors assigned budgets for time to complete task, number of CPUs, and memory in CPU, based on human work. Surprisingly, we found these estimates too tight in several cases: the verifier itself did not have headroom to run after the agent's solution completed, producing failures that depended on the agent's resource use rather than its correctness. We note that this is also a consequene of benchmark drift. Furthermore, we observed that different VM providers (Docker, Daytona) 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 a $1.5\times$ slack on the binding axis, restoring a feasibility floor.

The extended per-task taxonomy with secondary categories appears in Appx. A.3 (Table 4). 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 Sec. 4.

4. Results

Sec. 4.1 shows the TB Audit lifts pass rates while keeping ranks highly correlated. Sec. 4.2 shows the diversity of per-task compute as a separate scaling axis. Sec. 4.3 shows within-vendor harness sensitivity.

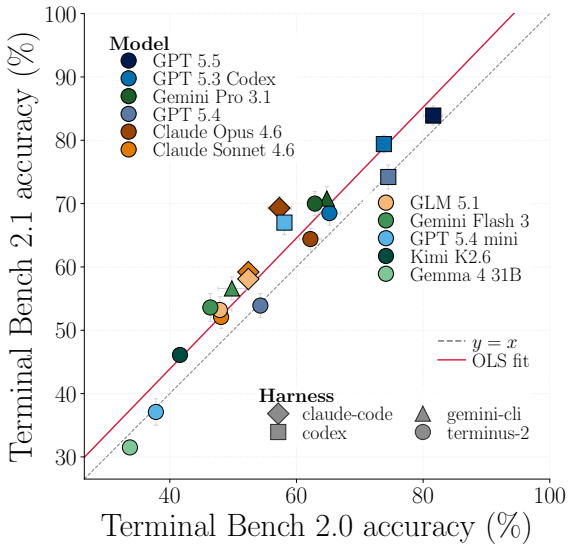

Figure 4. **Terminal Bench 2.1 keeps pair ranks highly correlated.** Spearman $\rho = 0.958$ with a task-bootstrap 95% CI of [0.881,0.979] across 19 matched agents.

4.1. Audit lifts pass rates without reordering agents

The audit leaves leaderboard agent ranking stable. Across 19 agents, the correlation in accuracy from TB 2.0 and TB 2.1 evaluations is high (Spearman $\rho = 0.958$, 95% CI $[0.881, 0.979]$; Fig. 4); the OLS fit is TB2.1 $= 2.6 + 1.03$TB2.0, and the mean score shift is $+4.4$ pp. 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.

After the audit, five tasks moved from $\leq 15.8\%$ to $\geq 56.1\%$ pass rate after verifier fixes alone, accounting for $\sim 62.5\%$ of the total absolute pass-rate change on the modified subset (Table 8; full derivation in Appx. C.1). These are tasks in the specification–verification mismatch category (Fig. 3a). The remaining 23 modified tasks contribute smaller deltas in both directions. Some asks changed difficulty after the audit, and thus had lower pass rates in TB 2.1.

4.2. The audit shifts cost-aware coverage curves

Timeouts dominate failures for low-performing agents (Fig. 3(right)), so a single pass-rate at a fixed budget conflates capability with budget tightness. We instead measure how much wall-clock or token budget is required to solve a task. Fig. 5 plots the budgeted coverage (Fan et al., 2025): For each agent and task, we find the cheapest successful trial. Sweeping a budget threshold over these per-task costs gives the curve. We sweep over time given to agents to complete the task. We do not consider other axes such as number of cpus or memory.

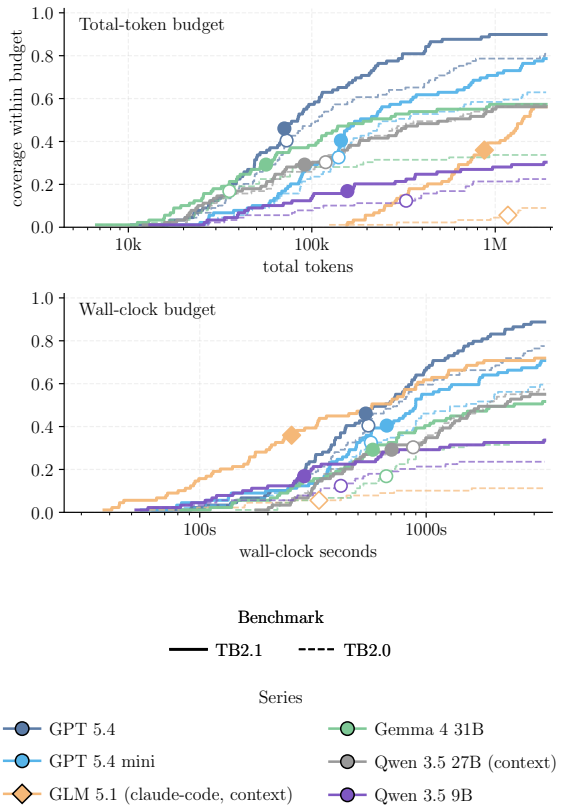

Figure 5. **Cost-aware effectiveness shifts across the audit.** For each agent, a task coverage is credited at the cheapest successful trial among full-benchmark timeout cells with timeouts $1\times, 4\times, 16\times$ *and* $32\times$. Curves are empirical step functions over tasks. GLM 5.1 and Qwen 3.5 27B have trials with timeouts $1\times, 32\times$, so we do not interpret their apparent slopes. The plotted curves are the coverage-within-budget functions, and the budget caps are the 95th percentile of finite successful-task costs across the plotted pairs.

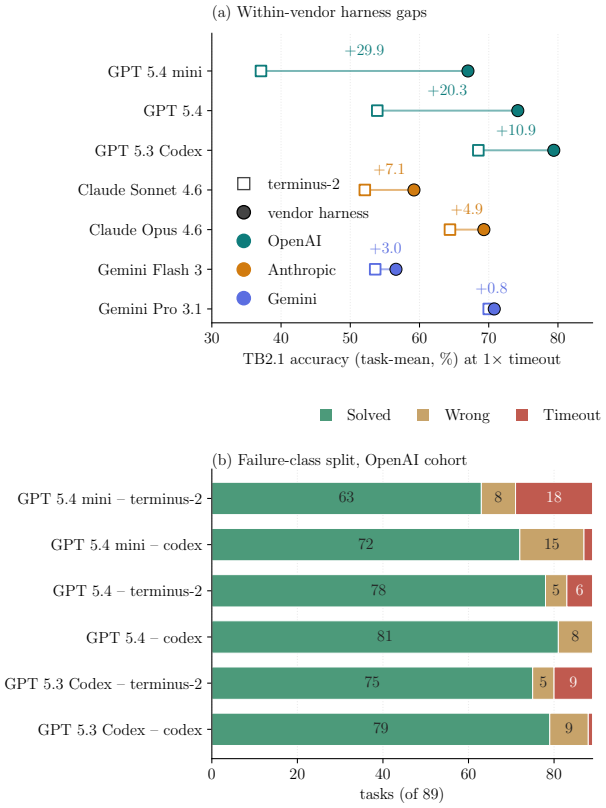

Figure 6. **Within-vendor harness gaps are a timeout effect on the OpenAI cohort.** (a) Per-agent performance between preferred and base harness. (b) Failure-class decomposition for the three OpenAI models in (a): stacked task counts (out of 89) split into solved, wrong-answer, and timeout-bound. Across every OpenAI model, `terminus-2` produces more timeout-bound tasks than `codex` and fewer wrong-answer tasks, the harness gap is wall-clock-limited, not capability-limited. See Sec. 4.3.

Coverage gains are budget-dependent. Larger models are generally more resource-efficient than smaller ones (as expected), but the audit shifts coverage curves unevenly across agents, as shown in Fig. 5. For GPT 5.4 and GPT 5.4 mini using `terminus-2`, the TB2.1 curves dominate TB2.0: at comparable token or time budgets, the audited benchmark credits more tasks. GLM 5.1 shows a large TB2.1 shift in wall-clock coverage, but a smaller shift in total-token budget, we suspect this is because its trajectories carry large input/cache-token counts per turn. Qwen 3.5 9B and Qwen 3.5 27B are unchanged across benchmarks, suggesting their failures are not on the environment side. The largest frontier shifts appear for Gemma 4 31B and GLM 5.1.

The compute-scaling sweep above varied per-task budget at fixed harness. Next we vary harness at fixed budget.

4.3. OpenAI's harness gap is mostly timeout-bound

We compare `terminus-2` against each vendor's preferred harness: OpenAI on `codex`, Anthropic on `claude-code`, Google on `gemini-cli`. The magnitude of the harness swap differs by an order of magnitude across cohorts (Fig. 6). Every OpenAI model loses 10–30 pp moving from `codex` to `terminus-2`, with GPT 5.4 mini falling 29.9 pp ($67.0\% \rightarrow 37.1\%$); Anthropic and Gemini models lose at most 7.1 pp under the same swap, with Gemini Pro 3.1 essentially unchanged ($+0.8$ pp). Within the Anthropic and Gemini cohorts, within-vendor model gaps exceed within-vendor harness gaps; within OpenAI's cohort the ordering appears to invert.

The gap is wall-clock-bound, not capability-bound. Decomposing failures by class explains the asymmetry. `terminus-2` produces more timeout-bound tasks than `codex` across all three OpenAI models and *fewer* wrong-answer tasks. When `terminus-2` finishes a task it is

at least as accurate as `codex`; it just finishes less often within the default wall-clock cap. The harness gap reflects efficiency (wall-clock budget) sensitivity, not capability.

Running `terminus-2` + GPT 5.4 mini with highest time budget (multiplier of 16, i.e. until it can complete a task) reaches 65.2% accuracy, closing most of the 29.9 pp gap to `codex` + GPT 5.4 mini at the base time budget (67.0%). At a halfway point ($t_{\text{mult}} = 4$) `terminus-2` + GPT 5.4 reaches 75.3%, already exceeding `codex` + GPT 5.4 at $t_{\text{mult}} = 1$ (74.2%). The OpenAI within-vendor harness advantage documented by prior work (Kapoor et al., 2025; Merrill et al., 2026) which has been measured at fixed wall-clock budget, shrinks at larger budgets. Harness comparisons at fixed budgets conflate efficiency with capability. Further separating `codex`'s efficiency advantage into its possible sources such as process parallelism, episode caps, tool affordances, and prompt formatting is left to future work.

5. Discussion

Continuous validation. Our audit pipeline surfaced 28 task-side defects in TB2.0, including 12 cases of post-release drift in external dependencies. 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. Efforts like SWE-Bench Verified and WebArena Verified address defects at release time but imply a terminal validated state that does not exist for benchmarks tied to live infrastructure. 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.

Rank preservation under audit. Across 19 agents, max $|\Delta\text{rank}| = 3$, with 13 of 19 agents holding rank within ± 1 position (Spearman $\rho = 0.958$, 95% CI $[0.881, 0.979]$). Agent-level score shifts have median 5.3 pp (IQR 2.3–6.8). The largest gains concentrate on specification–verification mismatches, which were disproportionately load-bearing for the larger models in our cohort. TB2.0 numbers in technical reports thus remain valid as rankings, and can be read as lower bounds on absolute performance under the TB2.1 verifiers.

Efficiency is conflated with capability at fixed budgets. Within OpenAI models on TB2.1, switching from `codex` to `terminus-2` drops accuracy by 10.9–29.9 pp at the default wall-clock budget. Failure-class decomposition shows the gap is timeout-bound: `terminus-2` produces more agent timeouts and fewer wrong-answer trials than `codex` on the same model. Raising the per-task time budget to $16\times$ closes most of the gap. This means harness comparisons reported at a fixed budget measure the joint quantity (capability $\times$ scaffold efficiency at that budget), not capability alone, and the same model under different budget configurations can yield score differences comparable to a model upgrade.

Overall, this work provides (i) an audit methodology that surfaces and repairs benchmark defects while preserving coverage; (ii) the TB2.1 artifact, demonstrating that audit-driven repairs 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. Future work should generalize the audit methodology to other benchmarks.

References

Aleithan, R., Xie, H., Lo, D., and Wang, S. 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. 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.

Chan, J. S., Chowdhury, N., Jaffe, O., Aung, J., Sherburn, D., Mays, E., et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.07095.

Chen, M., Tworek, J., Jun, H., Yuan, Q., et al. Evaluating large language models trained on code. *arXiv preprint*, 2021. URL https://arxiv.org/abs/2107.03374.

Cohen, D. et al. Preference leakage: How llm-as-a-judge learns to favour the few, 2025. URL https://arxiv.org/abs/2502.01534.

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.

Fan, Z., Vasilevski, K., Lin, D., Chen, B., Chen, Y., Zhong, Z., et al. Swe-effi: Re-evaluating software ai agent system effectiveness under resource constraints. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2509.09853.

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: A family of highly capable multimodal models, 2024. URL https://arxiv.org/abs/2312.11805.

Google DeepMind. Gemma 4: Open multimodal models. Model card and announcement; https://ai.google.dev/gemma/docs/core/model_card_4, 2026. 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.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., 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.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. 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.

Kapoor, S., Stroebl, B., Kirgis, P., Nadgir, N., Siegel, Z. S., Wei, B., Xue, T., Chen, Z., et al. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2510.11977.

Laude Institute. 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.

Laude Institute. 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.

Liu, T., Wang, Z., Miao, J., Hsu, I., Yan, J., Chen, J., Han, R., Xu, F., Chen, Y., Jiang, K., et al. Budget-aware tool-use enables effective agent scaling. *arXiv preprint arXiv:2511.17006*, 2025.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., et al. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2308.03688.

Liu, Y., Zhu, Y., Xu, C., Zhang, J., Liu, J., Yu, D., et al. G-eval: Nlg evaluation using gpt-4 with better human alignment, 2023. URL https://arxiv.org/abs/2303.16634.

Meng, K., Huang, V., Steinhardt, J., and Schwettmann, S. Introducing docent, 2025. URL https://transluce.org/introducing-docent. Transluce AI Technical Report.

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

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. GAIA: A benchmark for general AI assistants, 2023. URL https://arxiv.org/abs/2311.12983.

Moonshot AI. Kimi K2: Open agentic intelligence. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2507.20534. 1.04T-parameter MoE; 32B active. Open-weight agentic coding model. We use Kimi K2.6 in the broader 20-pair audit pool.

Moonshot AI. Kimi K2 Thinking model card. Hugging Face, 2026. URL https://huggingface.co/moonshotai/Kimi-K2-Thinking. Terminal-Bench reported with the default Terminus-2 framework and a custom JSON parser; other coding tasks reported on an in-house harness derived from SWE-agent. All scores averaged over 5 independent runs.

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

Plesner, A., Guzmán, D., and Athalye, A. An imperfect verifier is good enough: Learning with noisy rewards. *arXiv preprint*, 2026. URL https://arxiv.org/abs/2604.07666.

Qwen Team. Qwen technical report, 2024. URL https://arxiv.org/abs/2309.16609.

Qwen Team. Qwen3.6 model card. Hugging Face, 2026. 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).

Recht, B., Roelofs, R., Schmidt, L., and Shankar, V. Do ImageNet classifiers generalize to ImageNet? *Proceedings of ICML*, 2019. URL https://arxiv.org/abs/1902.10811.

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

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).

Shankar, S., Zamfirescu-Pereira, J. D., Li, C., Piorkowski, D., Correll, M., Hullman, J., et al. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences, 2024. URL https://arxiv.org/abs/2404.12272.

SWE-agent Team. Mini swe agent. Code repository, 2025. URL https://github.com/SWE-agent/mini-swe-agent.

Thakur, A. et al. Judging llm judges, 2024. URL https://arxiv.org/abs/2406.12624.

Wang, X., Chen, B., Adelt, Z., Shi, Y., Liu, B., Yang, S., and Neubig, G. Openhands: An open platform for AI software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent:

Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2405.15793.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.03629.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2406.12045.

Yu, B. 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.

Zeng, A., Lv, X., Hou, Z., Du, Z., Zheng, Q., Chen, B., Yin, D., et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint*, 2026. URL https://arxiv.org/abs/2602.15763.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2306.05685.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., 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.

Zhu, Y. and Kang, D. Noisy data is destructive to reinforcement learning with verifiable rewards. *arXiv preprint*, 2026. URL https://arxiv.org/abs/2603.16140.

Zhu, Y., Jin, T., Pruksachatkun, Y., Zhang, A., Liu, S., Cui, S., et al. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2507.02825.

Appendix Contents

This appendix is organized around the paper's narrative. Appx. A documents the audit pipeline that produced TB2.1. Appx. B documents the evaluation setup used throughout the paper, models, harnesses, configurations, and per-agent deltas. Appx. C collects supplementary empirical results that complement Sec. 4.

A. Audit pipeline details

A.1. Continuous-validation results

The TB-Audit pipeline consists of two GitHub Actions workflows that run on every PR (Fig. 2). `/audit` runs LM-judges over the verifier source and trial logs. `/oracle` executes the reference solution on the current image and force-rebuilds when needed. Together, they cover the three failure categories of Sec. 3.2:

Failure category	Detected by	Decision basis
Benchmark drift	`/oracle`	force-build of reference solution
Specification–verification mismatch	`/audit`	verifier source
Resource mismatch	`/audit` and `/oracle`	`task.toml` caps and measured runtime

`/oracle` results. a PR triggers an `/oracle` on a pull request, and a workflow force-builds the task environment and runs the reference solution, verifying that a known-correct solution still passes. The cost is free per PR on public repos (the workflow runs on free `ubuntu-latest` runners).

`/audit` results. Triggering `/audit` calls the pipeline on the benchmark data, where LLM judges make a decision. Any judge or judges can be setup for this. We found that three judges facilitated this process.

Propose-only. Neither workflow can git-merge the PR; both post a comment and stop, with a maximum iteration of 5.. The `permissions:` block in each workflow grants `pull-requests: write` but not `contents: write`. Diffs for the maintainer to apply land in a shared document `audit/changes/<task>.diff`.

A.2. Trial pool for audit evidence (Dataset 1)

The audit pipeline (Sec. 3) ingests external evidence on TB2.0 from different sources:

(i) **`leaderboard data`**: the public Terminal-Bench leaderboard data.

(ii) **`web data`**: data the agent collects from the web.

A.3. LM-judge prompt

TB-Audit can run with any judge(s). An LLM judge receive the same input and the same prompt. For each task, the chosen judge receives the task specification, plus a pooled trial summary. It emits a single JSON object: `task`, `verdict`, `categories` (any non-empty subset of the three categories in Sec. 3.2), `confidence`, and `justification`.

When the three judges proposes a fix, the reconciliation stage, `proposed_changes`, classifies the proposal as (load-bearing vs. incidental test relaxation, instruction-edit-contradiction vs. clarification-only, resource-bump, environment-fix) and proposes a Patch in `audit/changes/<task>.diff`.

Setup. We ran the audit on all 89 tasks in TB2.0. We employ three judges from two providers: `gpt-4o-mini`, `gpt-4o`, `gemini-2.5-flash` (via the OpenAI-compatible endpoint, internal thinking disabled). Same prompt, same evidence, same per-task input. Input per task: the task spec (`instruction.md`, `task.toml`, `Dockerfile`, `tests/`, `solution/`) plus Dataset 1.

Operating points and cost. Ground truth data (collected by three reviewers, and checked reviewing committee) was available for a subset (15 problems) of the dataset, where the authors knew there were issues from PR requests.

As mentioned previously, this pipeline can operate with one or multiple judges. A single LLM judge compared against the ground truth data achieved ok performance at low cost. To improve over individual judge biases, we use cross-provider agreement, pairing an OpenAI judge with the Google judge. On the binary flag (flagged vs. `clean`), the cross-provider Cohen's κ reaches 0.77. Per category, cross-provider reached $\kappa = 0.60$ on specification–verification mismatch and $\kappa = 0.59$ on resource mismatch; benchmark drift remains noisier ($\kappa = 0.21$–0.41), consistent with its being the heterogeneous union of multiple distinct failure modes (missing-dep, unpinned-source, stale-fact, provider-variance). However, the `/oracle` workflow makes up for judges not been good estimators of resource issues.

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.

Judge A	Judge 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 category Table 4 shows the primary failure model assigned to each task found to be broken as determined by the TB-Audit. Table 3 shows the final set of 28 revised tasks, and which component in the task was updated for TB 2.1

Per-task change taxonomy. Table 4 shows every repair category assigned to each changed task; 8 of the 28 tasks appear in more than one column. The full per-task taxonomy is in the data companion.

Table 2. **TB-Audit operating points.** Six configurations of the same 8-code detection prompt, scored against the final TB2.1 release-diff labels (28 positives of 89 tasks). Prompt selection and judge-pair choice were tuned on a 15-task calibration subset. The single-judge / single-submission baseline (A) has moderate precision and recall; pooling trial logs across 4 submissions of varying strength (B) recovers every PR #53 positive (recall 100%) but flags many additional tasks the maintainers did not change (FPR = 88.5%). Consensus over two judges (C) is the highest-F1 single-shot configuration; the strong-codes filter (F) is the highest-precision. No single row dominates: a continuous-validation pipeline that surfaces every potentially-broken task for human review should sit near (B) or (E); a maintenance pass with limited reviewer bandwidth should sit near (C) or (F).

Configuration	TP	FP	Precision	Recall	F1	FPR
(A) single judge, single submission	23	28	45.1%	82.1%	58.2%	45.9%
(B) single judge, pooled traces (4 subs)	28	54	34.1%	100.0%	50.9%	88.5%
(C) two judges, AND rule	20	15	57.1%	71.4%	63.5%	24.6%
(D) two judges, OR rule	25	39	39.1%	89.3%	54.3%	63.9%
(E) pooled ∩ any-single	25	36	41.0%	89.3%	56.2%	59.0%
(F) single, strong codes only	14	9	60.9%	50.0%	54.9%	14.8%

Table 3. **The 28 tasks revised in TB2.1, broken down by which stage of the task definition was fixed.** Each task has 5 components (instruction, environment, oracle, tests, resources); a checkmark indicates the stage where a failure was found and patched. Resources are split into time, memory, and CPU budgets. The CPU column covers any CPU-accounting fix, including a Dockerfile-internal one such as `compile-compcert`'s `nproc` wrapper, which makes container `nproc` report the cgroup limit rather than the host count, not just changes to the `cpus` cap in `task.toml`.

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				✓			

Table 4. **Failure-mode categorization for the 28 tasks revised in TB2.1.** Each task is labeled with every task-side root cause that contributed to its repair: a mismatch between the natural-language instruction and the verifier (*specification–verification*), an undersized resource budget (*resource mismatch*), or post-release rot in external dependencies (*benchmark drift*). 8 of the 28 tasks were repaired for more than one root cause and appear in multiple columns, so column counts sum to $36 > 28$; the per-task category set is in `data/failure_categories.json` (Appx. A.3).

Spec–verifier mismatch (16)	Resource mismatch (8)	Benchmark 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

B. Evaluation setup

B.1. Models evaluated

The analyzed cohorts use 13 model entries across six model providers (Table 5). Eleven of these appear in the 19 agents used for the TB2.0–TB2.1 rank comparison; Qwen 3.5 9B and Qwen 3.5 27B appear only in the compute-scaling analysis.

B.2. Harnesses evaluated

We use four scaffolds (Table 6): the Terminal-Bench 2.0 reference baseline `terminus-2` (Merrill et al., 2025), Anthropic's Claude Code CLI (Anthropic, 2025), OpenAI's Codex CLI (OpenAI, 2025a; 2026), and Google's Gemini CLI (Google, 2025).

B.3. Agent configuration overrides

Table 7 lists configuration overrides for analyzed agents at the default profile. Rows marked "default" run the harness-level defaults documented in Table 6 unchanged.

`terminus-2` triggers proactive summarization when 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 254K 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 `claude-code` harness manages its own context compaction internally and this knob does not apply.

Open-weight sampling overrides. The Gemma 4 31B and Kimi K2.6 rows in Table 7 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 5. **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, 2025b)	OpenAI	OpenAI API	400K
GPT 5.4 (OpenAI, 2025b)	OpenAI	OpenAI API	400K
GPT 5.4 mini (OpenAI, 2025b)	OpenAI	OpenAI API	400K
GPT 5.5 (OpenAI, 2025b)	OpenAI	OpenAI API	400K
Gemini Flash 3 (Google DeepMind, 2024)	Google	Google API	1M
Gemini Pro 3.1 (Google DeepMind, 2024)	Google	Google API	1M
Gemma 4 31B (Google DeepMind, 2026)	Google	Together.ai	256K *(131K)*
GLM 5.1 (Zeng et al., 2026)	Z.AI	Anthropic-shape	200K
Kimi K2.6 (Moonshot AI, 2025; 2026)	Moonshot AI	Together.ai	256K *(131K)*
Qwen 3.5 9B (Qwen Team, 2024; 2026)	Alibaba	Together.ai	254K *(131K)*
Qwen 3.5 27B (Qwen Team, 2024; 2026)	Alibaba	Together.ai	254K

Table 6. **Harnesses evaluated.** All four are off-the-shelf public scaffolds. "Scaffold" summarises the agent loop type; "Defaults" lists the harness-level knobs that matter for the rest of the paper. Agent overrides on top of these defaults are listed in Table 7.

Harness	Scaffold	Source / defaults
`terminus-2`	single-agent ReAct (Yao et al., 2023)	TB v2 reference baseline (Merrill et al., 2025; Laude Institute, 2026a); JSON-action parser, shell + file I/O tools. **Defaults:** `enable_summarize=True`, `proactive_summarization_threshold=8000` (tokens of accumulated context), `temperature=1.0` for thinking-capable models.
`claude-code`	vendor-managed agentic CLI	Anthropic Claude Code, headless (Anthropic, 2025); self-managed tool palette + context compaction. **Defaults in our runs:** `reasoning_effort=high`, `max_thinking_tokens` unset (Anthropic API default), adaptive thinking on.
`codex`	vendor-managed agentic CLI	OpenAI Codex CLI, non-interactive (OpenAI, 2025a; 2026); native Responses-API tool calls, managed context. **Defaults in our runs:** `reasoning_effort=xhigh`, `use_responses_api=True`.
`gemini-cli`	vendor-managed agentic CLI	Google Gemini CLI, Apache 2.0 (Google, 2025); shell + file I/O tools, managed context. **Defaults in our runs:** no harness-level overrides except those forced by the model route (Table 7).

Table 7. **Configuration overrides.** "Default" means the harness-level defaults of Table 6 apply unchanged. The open-weight rows (Gemma 4 31B, Kimi K2.6) pin `model_info`, raise the proactive-summarization threshold to 131K, and set sampling knobs (`top_p=0.95`, `top_k=20`); the full settings are listed in the prose preceding this table.

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 8 reports the per-task pass-rate change from TB2.0 to TB2.1, averaged across the version-matched cohort of 19 agents. Five tasks move from *unsolved* (TB2.0 pass rate $\leq$ 15.8%) to *solved* (TB2.1 pass rate $\geq$ 56.1%): `caffe-cifar-10` (5.3% → 70.2%, Δ = +64.9 pp), `polyglot-c-py` (5.3% → 84.2%, +78.9 pp), `polyglot-rust-c` (5.3% → 70.2%, +64.9 pp), `sam-cell-seg` (0.0% → 56.1%, +56.1 pp), and `torch-tensor-parallelism` (15.8% → 70.2%, +54.4 pp). All five were TB2.0 tasks where the verifier systematically rejected correct solutions; once the verifier is fixed, the underlying capability shows up.

Where the 62.5% comes from. Let $\Delta_t = \text{acc}_{\text{TB2.1}}(t) - \text{acc}_{\text{TB2.0}}(t)$ be the per-task pass-rate change averaged across the version-matched cohort, reported in Table 8 for each t in the 28-task changed subset $\mathcal{T}_{\text{changed}}$. Let $\mathcal{U} \subset \mathcal{T}_{\text{changed}}$ be the five unlock tasks listed above. Then $\sum_{t\in\mathcal{U}} \Delta_t = 319.2$ pp and $\sum_{t\in\mathcal{T}_{\text{changed}}} |\Delta_t| = 510.4$ pp, so the unlock share is $319.2/510.4 = \mathbf{62.5}\%$, the headline number quoted in Sec. 4.1.

Most agents gain on the changed-task subset (range +11.9 to +28.5 pp across the 19 agents reported in Table 9). Four agents regress on this subset: three OpenAI agents slip ∼1.1–1.2 pp each (`codex` + GPT 5.4 at −1.2 pp, `terminus-2` + GPT 5.4 at −1.1 pp, and `terminus-2` + GPT 5.4 mini at −1.2 pp), and `terminus-2` + Gemma 4 31B slips more substantially at −7.2 pp, consistent with TB2.0 verifiers being permissive in ways that benefited weaker agents at the margin; once tightened, those gains evaporated.

Table 8. **Changed-task pass rates.** Rates aggregate over the version-matched cohort in our evaluation. Eighteen tasks gain (range +1.7 to +78.9 pp), one is flat, and nine regress; the largest drops are `overfull-hbox` (−19.3 pp), `query-optimize` (−7.1 pp), and `protein-assembly` (−5.3 pp).

Task	TB2.0	TB2.1
adaptive-rejection-sampler	33.3%	61.4%
build-pmars	73.7%	86.0%
build-pov-ray	73.7%	71.9%
caffe-cifar-10	5.3%	70.2%
compile-compcert	47.4%	50.9%
configure-git-webserver	43.9%	52.6%
crack-7z-hash	78.9%	86.0%
extract-moves-from-video	8.8%	14.0%
filter-js-from-html	0.0%	7.0%
financial-document-processor	64.9%	59.7%
fix-git	98.2%	94.7%
gpt2-codegolf	15.8%	19.3%
hf-model-inference	87.7%	86.0%
install-windows-3.11	0.0%	24.6%
make-doom-for-mips	7.0%	1.8%
mcmc-sampling-stan	73.7%	75.4%
mteb-leaderboard	29.8%	38.6%
mteb-retrieve	15.8%	43.9%
overfull-hbox	56.1%	36.8%
polyglot-c-py	5.3%	84.2%
polyglot-rust-c	5.3%	70.2%
protein-assembly	21.1%	15.8%
query-optimize	47.4%	40.3%
rstan-to-pystan	61.4%	61.4%
sam-cell-seg	0.0%	56.1%
torch-pipeline-parallelism	8.8%	10.5%
torch-tensor-parallelism	15.8%	70.2%
train-fasttext	5.3%	3.5%

The largest regression is `overfull-hbox` (−19.3 pp), but *not* because the TB2.1 verifier got stricter: the test files are byte-identical between versions. It is a benchmark-drift fix: the TB2.0 canonical solution had become unbuildable as Ubuntu archives dropped a `texlive-latex-base` pin, so the verifier had no working reference to compare agent outputs against. Re-enabling the build restored the reference and exposed wrong-answer trials that an absent-reference verifier had been letting through. The

Table 9. **Changed-task pass rates by agent.** Restricting attention to the revised subset isolates the effect of the fixes: most agents gain 14–28 pp here, while overall benchmark scores (Table 10) move only about 5 pp on average because the other 61 tasks are unchanged.

Agent	Model	TB2.0	TB2.1
codex	GPT 5.5	61.9% ± 3.1	81.0% ± 2.7
codex	GPT 5.3 Codex	46.4%	64.3% ± 3.4
terminus-2	Claude Opus 4.6	38.1% ± 2.7	60.7% ± 3.8
gemini-cli	Gemini Pro 3.1	39.3%	59.5% ± 3.8
codex	GPT 5.4	60.7%	59.5% ± 3.8
claude-code	Claude Opus 4.6	33.3% ± 3.6	58.3% ± 3.1
codex	GPT 5.4 mini	28.6%	57.1% ± 4.3
claude-code	Claude Sonnet 4.6	35.7%	57.1% ± 2.9
terminus-2	Gemini Pro 3.1	32.1%	53.6% ± 3.9
terminus-2	GPT 5.3 Codex	36.9% ± 3.4	51.2% ± 4.3
terminus-2	Claude Sonnet 4.6	39.3%	51.2% ± 3.1
gemini-cli	Gemini Flash 3	31.0% ± 4.1	48.8% ± 3.4
claude-code	GLM 5.1	33.3%	46.4% ± 3.1
terminus-2	GLM 5.1	26.2% ± 2.4	42.9% ± 3.8
terminus-2	Gemini Flash 3	17.9%	41.7% ± 4.1
terminus-2	Kimi K2.6	25.0%	39.3%
terminus-2	GPT 5.4	32.1%	31.0% ± 3.6
terminus-2	Gemma 4 31B	28.6%	21.4%
terminus-2	GPT 5.4 mini	21.4%	20.2% ± 4.1

Table 10. **Leaderboard accuracy.** Fixes raise scores for nearly every agent while keeping rank shifts small; the rank-stability statistic is reported with the matched scatter in Fig. 4. Intervals are cross-task standard errors when defined. The Δ column reports TB2.1 − TB2.0 in percentage points; Δrank reports $\text{rank}_{\text{TB2.1}} - \text{rank}_{\text{TB2.0}}$. Thirteen of 19 agents stay within ± 1 rank position, with max $|\Delta\text{rank}| = 3$; $|\Delta\text{rank}| \geq 2$ for `claude-code` + Claude Opus 4.6 (-3), `terminus-2` + GPT 5.3 Codex ($+3$), `terminus-2` + Claude Opus 4.6 ($+2$), `terminus-2` + GPT 5.4 ($+3$), `terminus-2` + Gemini Flash 3 (-2), and `terminus-2` + Claude Sonnet 4.6 ($+2$).

Agent	Model	TB2.0	TB2.1	Δ (pp)	Δrank
codex	GPT 5.5	81.6% ± 1.5	83.9% ± 1.5	+2.3	+0
codex	GPT 5.3 Codex	73.8%	79.4% ± 1.4	+5.6	-1
codex	GPT 5.4	74.5%	74.2% ± 1.9	-0.3	+1
gemini-cli	Gemini Pro 3.1	64.8%	70.8% ± 1.9	+6.0	-1
terminus-2	Gemini Pro 3.1	62.9%	70.0% ± 1.9	+7.1	-1
claude-code	Claude Opus 4.6	57.3% ± 1.9	69.3% ± 1.8	+12.0	-3
terminus-2	GPT 5.3 Codex	65.2% ± 1.8	68.5% ± 1.9	+3.3	+3
codex	GPT 5.4 mini	58.1%	67.0% ± 1.9	+8.9	+0
terminus-2	Claude Opus 4.6	62.2% ± 1.8	64.4% ± 1.7	+2.2	+2
claude-code	Claude Sonnet 4.6	52.4%	59.2%	+6.8	-1
claude-code	GLM 5.1	52.4%	58.1% ± 1.8	+5.7	+0
gemini-cli	Gemini Flash 3	49.8% ± 2.2	56.6% ± 1.8	+6.8	-1
terminus-2	GPT 5.4	54.3%	53.9% ± 1.9	-0.4	+3
terminus-2	Gemini Flash 3	46.4%	53.6% ± 2.2	+7.2	-2
terminus-2	GLM 5.1	47.9% ± 1.9	53.2% ± 2.1	+5.3	+0
terminus-2	Claude Sonnet 4.6	48.1%	52.1% ± 1.8	+4.0	+2
terminus-2	Kimi K2.6	41.6%	46.1%	+4.5	+0
terminus-2	GPT 5.4 mini	37.8%	37.1% ± 2.1	-0.7	+0
terminus-2	Gemma 4 31B	33.7%	31.5%	-2.2	+0

next largest drops are `query-optimize` (-7.1 pp) and `protein-assembly` (-5.3 pp).

C.2. Trial-level transition matrix

Trial-level transition matrix. The per-task lifts in Table 8 aggregate to a population-level rescue rate. For each (agent,task) cell on the 28 changed tasks at $1\times$, we bucket each trial as pass, fail, or error. TB2.0 and TB2.1 trials in the same cell are separate physical runs, not literal trial-level matches, so we treat them as exchangeable samples and compute the within-cell joint distribution as the outer product of the two empirical marginals, averaged with equal weight per cell across 532 cells (19 agents $\times$ 28 tasks). The conditional rates are in Table 11.

Table 11. **Trial transitions on changed tasks.** Within each (agent,task) cell on the 28 changed tasks at $1\times$, TB2.0 and TB2.1 trials are treated as independent samples from the same condition; the joint bucket distribution is the outer product of the two empirical marginals, averaged with equal weight per cell over the 532 cells and 19 agents with native trials on both versions. The table reports the conditional $P(\text{TB2.1} = j \mid \text{TB2.0} = i)$. Bucket definitions: *pass*: reward $= 1$; *fail*: clean verifier verdict, reward $\neq 1$; *error*: `AgentTimeoutError` or `VerifierTimeoutError`. Two rescue rates fall out: the verifier/spec-fix rescue, $P(\text{pass} \mid \text{TB2.0 fail}) = 43.2\%$ (TB2.0 ran cleanly but scored zero on a task whose verifier was later fixed), and the resource-fix rescue, $P(\text{pass} \mid \text{TB2.0 error}) = 19.1\%$ (TB2.0 hit a timeout or resource error; TB2.1 finishes and passes). Pooled, of TB2.0 trial-attempts that were not a clean pass, $\mathbf{32.7}\%$ are clean passes in the same cell on TB2.1. Weighting by provider compute spend instead of trial counts, of the 476M input + output tokens TB2.0 burned on non-pass trials on these tasks, $\mathbf{28.8}\%$ ($137M$ tokens) were spent on cells where TB2.1 demonstrates the task is actually solvable.

	TB2.1 bucket		
TB2.0 bucket	pass	fail	error
pass	0.748	0.128	0.124
fail	0.432	0.438	0.131
error	0.191	0.110	0.698

C.3. Failure composition by harness and tier

Fig. 7 decomposes every agent cell on TB2.1 at $1\times$ timeout into the three trial outcomes that remain after the infrastructure-error filter: *pass* (reward $= 1$), *clean wrong answer* (verifier returned a verdict, reward $\neq 1$), and *agent timeout* (`AgentTimeoutError`). The bottom band of agents (Qwen 3.5 9B, Gemma 4 31B, Kimi K2.6, and the weakest GPT 5.4 mini cells) is dominated by timeouts rather than wrong answers—those agents do not run out of capability so much as out of wall-clock to deploy it.

Failure mode shifts with capability. Fig. 8 pools Fig. 7 by per-agent pass rate (frontier $\geq 70\%$, mid 50–70%, small $< 50\%$). Timeout share grows monotonically across tiers ($7\% \to 24\% \to 56\%$) while wrong-answer share drifts down ($17\% \to 16\% \to 10\%$), quantifying the small-model timeout dominance noted in Sec. 4.2.

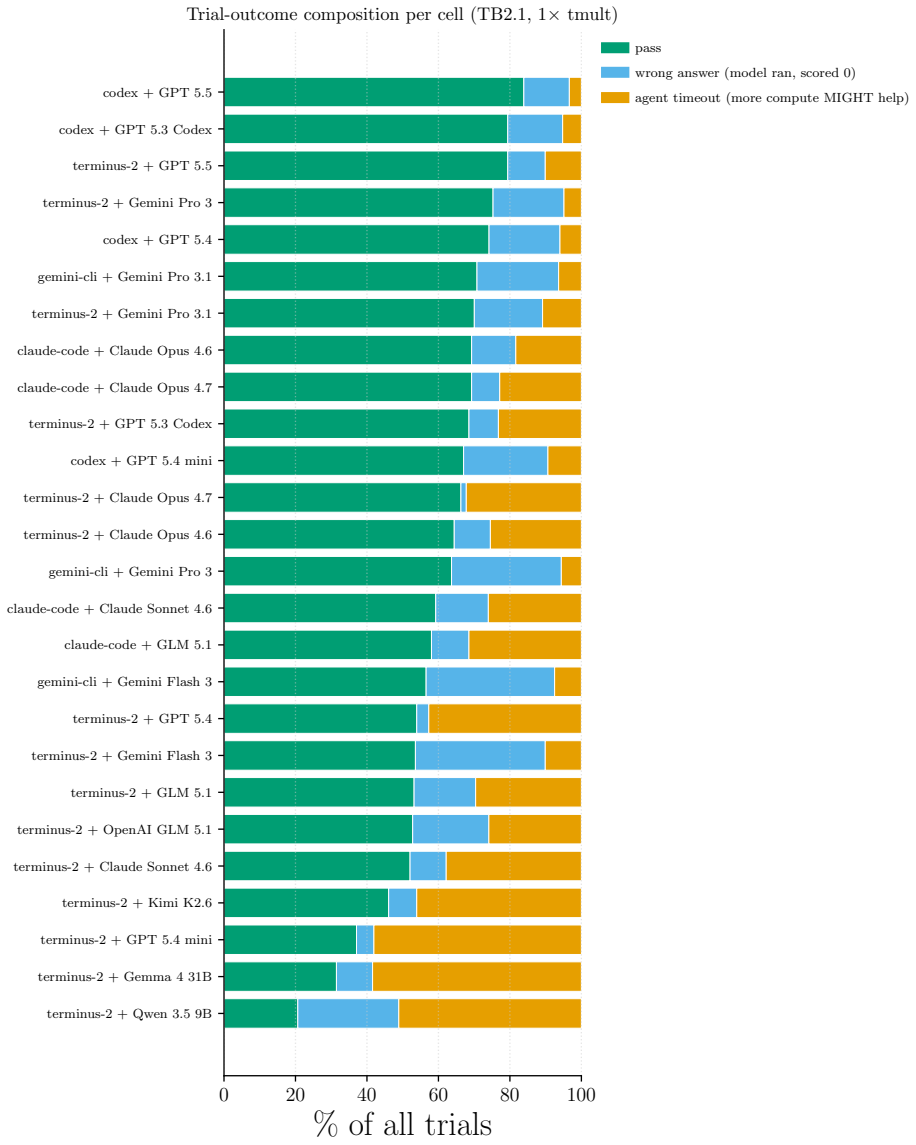

Figure 7. **Trial outcomes by agent.** Each horizontal bar shows, for one agent, the fraction of trials that passed (green), returned a clean wrong answer (sky blue), or hit `AgentTimeoutError` (amber). Pairs are sorted top-to-bottom by pass rate. *Filter disclosure:* infrastructure/operator failures (`NonZeroAgentExitCodeError`, `DaytonaError`, `CancelledError`, `AuthenticationError`, `RuntimeError`, `AgentSetupTimeoutError`, and similar non-model errors) are filtered upstream by `build_table.py`'s whitelist before this figure is computed. These *infrastructure-filter* trials account for ∼27.6% of upstream TB2.1 trials at 1× in the default-profile dump (computed live from `agent_model_bench.csv`'s `n_filtered_trials` column over the kept-trials total). They are excluded so the per-cell composition reflects model behavior rather than infrastructure noise. (Not to be confused with the trial-level *rescue rate* of 32.5% reported in Table 11, which is a different quantity: the fraction of TB2.0 non-pass attempts on the 28 changed tasks that flip to clean passes on TB2.1.)

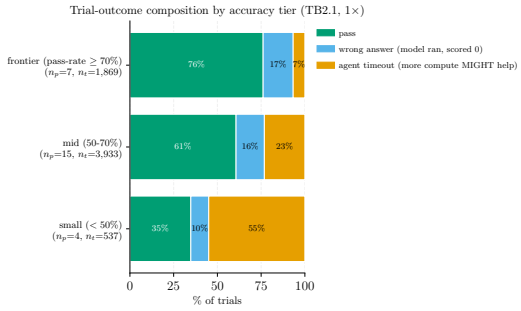

Figure 8. **Failure modes by capability tier.** Trials are pooled on TB 2.1 at $1\times$ timeout. Tiers are cut at the per-agent pass rate: frontier $\geq 70\%$ ($n_p = 4$ agents, $n_t = 1{,}068$ trials), mid 50–70% ($n_p = 12$, $n_t = 3{,}132$), small $< 50\%$ ($n_p = 4$, $n_t = 516$). The timeout share triples between frontier and mid ($7\% \rightarrow 24\%$) and again between mid and small ($24\% \rightarrow 56\%$); wrong-answer share drifts down monotonically. Infrastructure failures (`NonZeroAgentExitCodeError`, `DaytonaError`, `CancelledError`, etc.; $\sim$27.6% of upstream TB2.1 trials at $1\times$, distinct from the trial-level rescue rate of 32.5% in Table 11) are filtered by `build_table.py`'s whitelist before this figure is computed; see Fig. 7 for the per-agent breakdown.