Terminal-Bench Is Blind to Destruction

It grades what the task asked for, and nothing else the run touched.

June Kim
Independent Researcher
june@june.kim
ORCID 0009-0005-3153-9396

July 9, 2026

Abstract

Terminal-Bench grades a terminal agent by the properties of the final container state and never inspects the commands it ran, so it cannot see a side effect it never asserted against. It has the fail-to-pass half of SWE-bench's contract but lacks the pass-to-pass frame, the condition that a passing run leaves intact everything it was not asked about. We measure the gap with a model-free probe, run over all 89 tasks with a re-runnable receipt each: compose the task's own reference solution with a careless suffix, a documented terminal-agent accident such as `rm -rf .git`, re-run the official grader, and read the verdict. On Terminal-Bench 2.1, 40 of 83 gold-passing tasks (48%) certify success after a careless deletion inside the task's own workspace. Off the workspace, all 83 certify success after the run deletes user assets planted before the agent starts, a second repository, an SSH key, a customer-data file, and the blindness is structural, because a grader that reads only final state has nothing left to catch. The reward scores a destructive completion identically to a clean one. The 26-task 2.1 revision left the frame open on every task, and a dual defect sits on the specification side, where the instruction does not always pin what the grader checks. The cheap, drop-in fix gates the agent's state delta, diffed from a pre-agent snapshot, against the oracle's own footprint.

1 Introduction

Why do so many developers still refuse to run a terminal agent unattended? A failed task does not explain it: that cost is small, you notice, and you do it by hand. The answer is the stories. An agent that resets a repository and loses a day of uncommitted work, that drops a database, that deletes the wrong directory. The dominant risk in a terminal is that the work gets done and something else gets destroyed on the way.

The reluctance is documented, and it is specific. In April 2026 a Cursor agent, sent to fix a staging bug, found an unscoped infrastructure token and used it to delete a company's production database and its backups in nine seconds, with no confirmation step. The agent's own postmortem admits it guessed at the scope of a destructive call instead of verifying it (The Register, 2026). It is not an isolated anecdote. The first systematic study of agent filesystem misuse gathers 290 such reports across 13 frameworks and summarizes the pattern in a line: coding agents "regularly corrupt data, delete files, and leak secrets" (Zhong et al., 2026).

The tooling ecosystem has converged on one guardrail in response. Anthropic, GitHub, and OpenAI all gate the same actions, deleting files, dropping tables, `rm -rf`, `git push --force`, `git reset --hard`, behind explicit confirmation, because those are the actions you cannot take back. Developers have drawn the obvious conclusion: in the 2025 Stack Overflow survey a majority still do not run agents autonomously, and 81% report security or privacy concerns about the agents they do use. The gate on an unattended terminal agent is whether it can be trusted not to wreck the machine doing the work.

A terminal-agent benchmark is read as evidence of deployment readiness, and Terminal-Bench (Shaw et al., 2026) is the one that reading runs through: a set of containerized tasks, each with an instruction, tests over the final container state, and a human-written reference solution, with a public leaderboard the labs report against. Its authors state the aim plainly, to measure whether agents can do the high-skill terminal work professionals are paid for, and they build the tasks to reflect real deployment rather than a lab exercise. We take that goal at its word. Real deployment is exactly where the destruction happens, so it is what a passing grade should rule out. An agent that finishes the task and deletes the repository is a real deployment outcome, and Terminal-Bench passes it. The benchmark grades what the agent accomplished. Here we measure what it grades away.

One objection arrives before any evidence: Terminal-Bench is a capability benchmark, and safety is somebody else's job. There is a real safety literature, and it does measure destructive behavior, but it lives in separate red-team benchmarks that a model is scored on once and is never trained against. Capability is scored on the board that drives development, and that board is Terminal-Bench. Safety measured off the leaderboard puts no pressure on the leaderboard.

So a benchmark that ranks terminal agents while its grader cannot see destruction is scoring the half that is easy to score and letting that stand for competence. The gap this audit measures is therefore the safety term missing from the capability score everyone reads.

Terminal-Bench's tests are outcome-driven. They "do not test the agent's commands or console output"; they check properties of the container after the agent stops. That is a good decision for measuring accomplishment, and a blind spot for measuring harm, because harm is mostly a property of what the agent touched that the task never mentioned. A test that confirms a certificate was generated says nothing about whether the home directory survived. The grader looks where the task points, and the destruction happens where it does not.

We measure the size of that blind spot. The method needs no model and no judgment: take each task's own reference solution, the one the authors certified solves it, and append a single careless action. Then run the benchmark's own grader and read its verdict. If a solution that also deleted the repository still passes, the benchmark cannot tell a clean run from a destructive one, and its score is not a claim about safety at all.

2 The missing frame condition

The gap has a name in SWE-bench, the benchmark whose problem shape Terminal-Bench shares. SWE-bench grades a patch on two test sets: FAIL_TO_PASS, the tests that must go from failing to passing, which is the fix, and PASS_TO_PASS, the tests that passed before and must still pass after. PASS_TO_PASS is the frame: the guarantee that the patch solved its problem without breaking everything around it. Remove it and a patch that fixes the bug by deleting half the module scores the same as a clean one.

The name is older than SWE-bench. In program verification a specification's frame, or modifies-clause, is the state an operation may change, and by implication what it must leave untouched (Reynolds, 2002); it is the executable form of McCarthy and Hayes's frame problem (1969). A benchmark with no frame condition scores as if every task's modifies-clause were the whole machine.

Terminal-Bench has the fail-to-pass half. Its outcome tests are exactly the check that the task went from undone to done. It has nothing playing the pass-to-pass role. There is no set of properties a passing solution must leave undisturbed, so the container outside the graded properties is a free surface. Delete anything there and the score does not move.

> Picture the final container state as one large box. The grader asserts only a small region of it, the graded target. The rest is the frame, state the task never named, which the grader never reads. A destructive action (`rm -rf .git`; the other repo, the SSH key, the data: gone) lands in the frame, and the reward is still 1: destruction in the frame is unasserted, so the score does not move.

This is not an oversight that more care removes, and the reason is worth stating, because it is the objection every reader raises first. If the authors can afford to hand-write 89 tasks, surely they can afford to hand-write graders that check for bad side effects. They can afford the labor. The labor is not the problem.

The target of a task is a thing you specify: generate a cert, merge the branch. The frame is the complement of that thing: everything else in the container that must stay as it was. You cannot hand-enumerate that complement to completeness. To write a frame check by hand you would assert the preservation of the whole pre-existing world, every file, the git history, the running services, the user's other data, most of which the

task never names because it is not what the task is about. The target is finite and you write it down; its complement over a full Linux container is unbounded, and you cannot.

Terminal-Bench's own corpus proves this is the mechanism rather than negligence. One task, `sanitize-git-repo`, does hand-author a frame check: `test_no_other_files_changed`, a diff against a pinned commit. The authors wrote it in the one place the frame was small enough to enumerate, a git repository, where "everything else" collapses to a single `git diff`. It is the one task where a frame check was tractable, and it is the one task that has one. Where the frame is the whole filesystem, no such check appears. That is a tractability pattern.

3 Method

For each task we drive the benchmark's own artifacts directly, the same ones the official runner drives, so a skeptic reproduces a verdict without trusting our harness.

The steps, per task and per mutation:

1. Pull the task's pinned image, recorded in its `task.toml` as `docker_image`, and record its resolved digest so the artifact is fixed even if the tag moves.
2. Boot a container under the task's own resource caps (`cpus`, `memory_mb`).
3. Run the reference solution, `solution/solve.sh`, staged exactly as shipped. This is the baseline, and a baseline that fails is a separate finding: the gold solution does not pass its own grader.
4. Snapshot the container filesystem with `docker diff`. This is the oracle footprint: the changes the reference solution legitimately made, each recorded as a path and a kind, added, modified, or deleted. A path the oracle modifies is not a license to delete it.
5. Append one careless mutation, applied only where it structurally can be.
6. Run the official grader, `tests/test.sh`, which writes `/logs/verifier/reward.txt`. Read the reward.

A mutation is careless. It is a documented terminal-agent accident, the class the introduction's incident reports record in the wild. The run is the oracle's own solution plus that accident, so there is no exploit agent anywhere in the pipeline. The suite:

mutation	action	applies when
`nuke-git`	`rm -rf .git` in the working tree	tree has a `.git`
`reset-hard`	`git reset --hard HEAD~3`	tree has a `.git` with history
`nuke-preexisting`	delete pre-existing workspace files not in the oracle footprint	any untouched pre-existing file
`wipe-sentinel`	delete planted off-task user assets (a second repo, an SSH key, a data file)	any task

A destructive mutation that survives, reward still 1, is frame-blind: the grader certified success for a trajectory that made a change the reference solution never made, most often a deletion the reference never performed. A mutation that is caught, reward 0, proves the grader can see damage when the damage reaches a property it asserts. The contrast between the two is the whole finding.

Every run persists its image digest, the oracle footprint, the list of files the mutation deleted, the full grader output, and the reward. The verdict re-derives from those without rerunning anything.

4 Results

On Terminal-Bench 2.1, over all 89 tasks.

4.1 Baseline: the gold-passing rate

The reference solutions mostly pass, and six do not. Baseline gold-passes: 83 of 89. Six reference solutions fail their own grader under a clean rerun (`query-optimize`, `crack-7z-hash`, `sqlite-with-gcov`, `build-cython-ext`, `caffe-cifar-10`, `compile-compcert`). Three are heavyweight build and training tasks where the failure is consistent with our harness's resource ceiling rather than a defect, and we hold them as unclassified pending a per-task check. We quarantine all six rather than adjudicate them here: they are excluded from the 83-task denominator, and two of them are among the rerun-unstable tasks noted below. None is asserted as a finding.

4.2 Frame-blindness across mutations

Over the 83 gold-passing tasks:

mutation	frame-blind (survived)	caught	not applicable
`nuke-preexisting`	35	26	22
`nuke-git`	6	3	74
`reset-hard`	6	3	74

40 of 83 gold-passing tasks (48%) survive at least one destructive accident the reference solution never performed. The 40 is the union over the three mutations, overlap removed. The 26 tasks that catch `nuke-preexisting` are the control: their graders do read the files, so a survivor is genuine blindness. Indiscriminate destruction would have been caught. The frame-blind set is not a corner of the benchmark; it spans forensics (`git-leak-recovery`, `password-recovery`, `extract-elf`), data pipelines (`reshard-c4-data`, `portfolio-optimization`), machine learning (`train-fasttext`, `protein-assembly`), long-running services (`qemu-startup`, `mailman`), and the plainest specimen, `fix-git`, which asks the agent to merge a branch and scores a pass after `solve.sh` followed by `rm -rf .git`. The whole repository is gone; the merge is certified complete.

4.3 Destruction of unambiguously off-task state

The defect is at its cleanest when the destroyed state is unarguably not the task's. Before the solution runs we plant off-task user assets: a second git repository of unpushed work, an SSH private key, a customer-data file, all under one directory no task references. The mutation deletes only that directory. Every gold-passing task still passes: 83 of 83 certify success after the planted assets are wiped, none caught.

Unlike the workspace census, this outcome is entailed. Deleting assets planted after the image snapshot returns the container to a state indistinguishable from a clean run, so a grader that reads only final state has nothing left to catch, however carefully its tests are written. The 83 runs are the receipt that Terminal-Bench is such a grader on every task, with no global assertion hiding in any test or in the harness. The verdict is a function of a state slice that cannot encode off-task harm, whatever the agent did on the way; that is the finding in its strongest form. It also settles the objection that a deleted `nuke-preexisting` input might have been disposable: an SSH key and someone's other repository are not. The gate in the discussion catches exactly this class, because it diffs against the pre-agent world rather than reading the final state alone.

Rerun any row and check it without reading this prose: `regrade.sh` emits the task's `oracle_diff`, its `deleted` list, and the grader's own `reward.txt`. An audit is a rerun, so it has to be re-runnable to be an audit.

5 Persistence under the 2.1 revision

A benchmark that ships a fix is a moving target, so we audit the version that was already hardened. Terminal-Bench 2.1 is a verified revision of 2.0. Drawing many changes from an independent verification effort, its authors modified 26 tasks across three axes: external dependencies that had drifted, resource and timeout budgets too tight for a valid solution, and misspecification, including `query-optimize`, whose tests had expected Spark SQL while its instruction asked for PostgreSQL.

Every one of those axes is a real construct-validity concern, and none of them is the frame. Determinacy, hermeticity, and resource budgets are about whether the target is well posed. The frame is about the complement. A careful re-verification pass touched 26 tasks and left the pass-to-pass hole open on all of them. The defect survives exactly the scrutiny that fixed everything around it, because there is no cheap source to derive it from. On the five 2.0 tasks we ran before moving to 2.1, the frame-blind verdicts matched their 2.1 counterparts; the complete task-by-task 2.0-to-2.1 delta is left to the harness, which regenerates it from a 2.0 clone.

6 The reward ordering

Set what a user wants against what the benchmark pays, and the asymmetry is the sharpest form of the finding:

outcome	what the user wants	reward
clean completion	best	1
safe failure (task undone, nothing broken)	acceptable, do it by hand	0
destructive completion (task done, data gone)	worst	1

The reward is binary over completion, so a destructive completion earns the same 1 as a clean one and a safe failure the same 0 as a reckless one. It places the outcome the user fears most, at 1, above the harmless failure at 0, and cannot separate the run they dread from the one they want.

We are auditors, so the claim is about the score. The grading can certify a destructive run as a full pass; whether a trainer or an agent exploits that is an implementation detail we do not measure. What the score does fix is a ceiling on what safety can earn here. A confirmation prompt that declines a risky action can only lose the tasks it blocks, never gain credit for the damage it prevented, because the grader carries no term for prevented damage. Safety that sometimes costs a completion cannot show up positive on this board.

7 The dual defect: specification determinacy

The frame is one of two ways a Terminal-Bench pass can mean less than it appears to. The frame hole is on the oracle side: the grader does not guard what the task never asked about. There is a matching hole on the specification side, where the instruction does not always pin the value the grader checks, so a solver who reads it faithfully and produces a different but reasonable output fails. Passing means recovering the author's unstated choice rather than solving the stated problem. The two are duals: one asks whether the materials pin what is graded, the other whether the grader guards what is not. `fix-git` sits on both. The grader forgives the deletion of the repository, and the merge it hashes is a real conflict: master's tip and the lost commit both edit `about.md`. The grader accepts only the resolution that discards master's newer edit wholesale, a choice the instruction never states.

Determinacy is the subject of a companion line of model-free audits on SWE-bench Pro and DeepSWE (Kim, 2026), which put a proven floor under each. A first-pass reading over Terminal-Bench 2.1 finds the specification side tighter here than on those mined-pull-request benchmarks, to the authors' credit. Eight candidates survive, one a temporal case where the instruction names a month ("as of August 2025") and the grader pins a single snapshot. We flag these as a cold read and leave the adjudicated floor to that companion instrument; the receipt-backed number in this audit is the frame census. The pairing is the point: two sides of one question, and Terminal-Bench answers neither.

8 Related work

The frame concept is not ours. SWE-bench's PASS_TO_PASS is the frame made routine, and it is routine there for a reason we lack here: the repository ships a test suite, so the frame is mined for free by picking the tests that pass at the base commit and still pass after the gold. Terminal-Bench has no pre-existing regression suite over container state, so the cheap source is gone. Our recommended fix, deriving the frame from the oracle's footprint, is the terminal-world substitute for that free mine.

The safety literature that measures destructive behavior is real and growing, and it sits apart from the capability leaderboard the labs optimize.

benchmark	what it measures	on the capability board?
ToolEmu (Ruan et al., 2024)	tool-use harm in an emulated sandbox; 68.8% of flagged failures are real risks	no
RedCode (2024)	risky code execution and generation, 25 vulnerability types	no
Agent-SafetyBench (Zhang et al., 2024)	agent safety across 349 environments; no agent scores above 60%	no
SABER (Hu et al., 2026)	operational safety from the final state of git-backed workspaces	no

The last column is the point. These are the right instruments, and the introduction gave the reason they do not close the gap: they are scored off the board that drives development. SABER is nearest to this audit, since it names the exact failure we exercise, a destructive side effect produced as a byproduct of pursuing a legitimate goal. The frame gate the discussion recommends puts a safety term back on the board itself.

The reward-hacking work on Terminal-Bench, including the authors' own hardening pass in 2.1, is adjacent but distinct. Reward hacking asks whether a wrong solution can pass. We ask the opposite: whether a right solution that also causes harm still passes. No adversarial agent appears in our pipeline, only the oracle's own solution plus a careless accident, so a finding here is about what a passing verdict certifies.

9 Discussion

Two axes, one principle: a benchmark's score is trustworthy when the materials pin the behavior it grades and the grader guards the state it does not. Terminal-Bench does neither reliably, and the frame hole is the one that turns a capability score into a safety hazard. The fixes worth asking for are the ones that drop in, because they are built from artifacts the benchmark already produces. None of these is a redesign.

1. Gate every task on the agent's state delta against the oracle's. The keystone, and it costs no per-task authoring. Manifest the filesystem at the moment the container is handed to the agent, one `find` at a point the harness already controls, and diff the final state against it; the reference solution run through the same two snapshots at build time yields the oracle's footprint, each entry a path and a kind. Add one harness-level check: the agent's delta must be subsumed by the oracle's, plus a declared tolerance, so no task scores 1 through a deletion or modification the reference never made. Deleting a path the oracle only modified is included, and that catches the plainest specimen, where the merge writes into `.git` and the accident deletes it. The reference point must be the pre-agent world. A diff against the image misses anything created and then destroyed between snapshot and verdict, exactly the shape of the sentinel wipe above; a pre-agent manifest cannot net it out. The tolerance is where a legitimate alternate solution declares its own footprint, so the gate is a conservative safety floor. It is not a correctness oracle. Report its pass rate next to completion and the score gains a safety term for free, since the verdict is already computed.
2. Pin or cut two underdetermined specifications. `mteb-leaderboard` grades a leaderboard "as of August 2025" with no snapshot shipped, and `query-optimize` grades a runtime bar against a golden the solver never sees. State the target in the instruction or drop the task. Two edits.
3. Repair or quarantine six failing golds. Six reference solutions fail their own grader under a clean rerun, two of them unstable across reruns. A gold that cannot pass its own test cannot anchor a verdict until the contradiction is resolved.

None of this finishes the job. The gate is a floor, because harm is adversarial: any fixed check becomes a target to route around, and a footprint diff sees only what it was built to see. It catches the single destructive step and misses the security-shaped tail: a secret read and sent over the network where no file changes at all, a value corrupted inside the allowed output, a weakened permission, an opened port.

SABER (Hu et al., 2026) marks where that line falls. It grades safety the same way, from the state delta a run leaves behind, but detects harm by matching the delta against a per-task catalogue of unsafe patterns plus an LLM judge, with no general preservation clause. The frame gate is that same delta check with the catalogue removed: it takes the oracle's own diff as the allowlist of what is permitted, which is why it is free and holds on every task at once. What it saves in cost it gives up in reach, missing the compositional harms SABER finds in roughly a third of its violating runs, where no single action is the whole of the damage. The two are a floor and a ceiling: the gate gives every task the preservation floor it lacks today, and SABER's per-task, judged evaluation is where the ceiling gets built.

Prevention is two-sided. The agent side owns judgment, the refusal and the confirmation before an irreversible action, and frameworks already invest there; OpenHands carries a whole apparatus for gating risky actions. The harness side owns measurement, and its absence is what this audit found. Neither closes the gap alone: a trusted-but-unmeasured agent is what practitioners already decline to run, and a measured harness with nothing to enforce selects for completion alone. They are meant to close a loop, the harness scoring harm and thereby selecting for agents that avoid it; by supplying no harness-side signal, Terminal-Bench leaves it open.

For anyone running or reporting a score, the reading is narrow. A Terminal-Bench number certifies task completion over the properties the grader checks, and nothing about what the run destroyed. Two agents at

the same score can differ without bound in what they wrecked getting there. Report completion and a safety metric apart, and do not read the leaderboard as a claim about which agent is safe to run unattended.

10 Threats to validity

- Harness fidelity. We replicate the runner rather than invoke it. The baseline gold-pass rate, 83 of 89, is our check on fidelity: where the reference solution passes, our staging matches the benchmark's closely enough to reproduce the intended verdict. The six baseline failures are quarantined pending classification. `query-optimize` merits the first look: 2.1 rewrote it, and a gold that fails its own rewritten grader is either an artifact of our run or a fresh defect.
- What counts as harm. `nuke-preexisting` deletes pre-existing files outside the oracle footprint, and for some tasks a deleted input may be disposable, so the 48% is a raw ceiling that a triage into clearly-valuable versus disposable state would lower. The `wipe-sentinel` tier removes the ambiguity about value (an SSH key is not disposable) at the price of being entailed rather than sampled: it shows the harm class is invisible to final-state grading in principle, while the workspace census is the measured rate. Weight the 48% as the measurement and the 83 of 83 as the structural limit.
- Determinism of the grader. A few tasks grade a live or timed property and can vary across runs. We cap each phase at a fixed wall-clock and note that the frame-blind verdicts concentrate in file-state tasks, where the rerun is stable.
- Scope. Findings are scoped to the public tasks of Terminal-Bench 2.1. The full 2.0 delta is in progress.

11 Conclusion

Terminal-Bench grades what a terminal agent accomplished and cannot see what it destroyed, because it has the fail-to-pass half of the contract and not the pass-to-pass frame. On the current version, half the gold-passing tasks survive a careless deletion inside their own workspace, and every one of them certifies success after the agent deletes planted user state it was never asked to touch, an outcome final-state grading entails and the runs confirm. The reward ordering scores a destructive completion level with a clean one and above a harmless failure. The fix for the gross case is cheap and derivable from the oracle the benchmark already ships. The rest is security work, split across the agent that must decline the harm and the harness that must score whether it did. A benchmark for effective terminal use has to measure the harm, because avoiding it is most of what "effective" means to the person deciding whether to let the agent run.

The whole audit, every per-task receipt, and the code that regenerates these tables are the artifact. This preprint is its readable surface.

Disclosure

We shared these findings with the Terminal-Bench corresponding authors on July 9, 2026, with a standing right of reply; any response will be linked here. The actionable recommendations are filed as terminal-bench#1459.

References

Shaw et al., 2026. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. arXiv:2601.11868. The benchmark under audit; its stated goals are quoted inline.

Terminal-Bench 2.1, 2026. Release notes. The 26-task verified revision this audit targets.

Jimenez et al., 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR 2024. arXiv:2310.06770. The FAIL_TO_PASS and PASS_TO_PASS frame contract.

Zhong et al., 2026. Don't Let AI Agents YOLO Your Files: Shifting Information and Control to Filesystems for Agent Safety and Autonomy. arXiv:2604.13536. The first systematic study of agent filesystem misuse: 290 reports across 13 frameworks.

The Register, 2026. Cursor-Opus agent snuffs out startup's production database. The PocketOS incident: production database and backups deleted in nine seconds by an autonomous agent with an unscoped token.

Stack Overflow, 2025. Developer Survey 2025. Agent-adoption figures: a majority do not run agents autonomously; 81% report security or privacy concerns.

McCarthy and Hayes, 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence 4. The frame problem.

Reynolds, 2002. Separation Logic: A Logic for Shared Mutable Data Structures. LICS 2002. The frame rule and the modifies-clause reading of the frame.

Ruan et al., 2024. Identifying the Risks of LM Agents with an LM-Emulated Sandbox (ToolEmu). ICLR 2024. arXiv:2309.15817. Agent harm in an emulated sandbox.

Hu et al., 2026. SABER: Benchmarking Operational Safety of LLM Coding Agents in Stateful Project Workspaces. arXiv:2606.01317. Grades safety from the final environment state after a sequence of actions in git-backed workspaces; the closest prior work to this audit.

RedCode, 2024. Risky Code Execution and Generation Benchmark for Code Agents. NeurIPS 2024 Datasets and Benchmarks. arXiv:2411.07781. 4,050 risky test cases across 25 vulnerability types.

Zhang et al., 2024. Agent-SafetyBench: Evaluating the Safety of LLM Agents. arXiv:2412.14470. 2,000 cases across 349 environments; no evaluated agent scores above 60%.

Kim, 2026. A Determinacy Audit of SWE-bench Pro; Auditing DeepSWE. The spec-side companion audits in this program.

LLM use

This work used a large language model (Claude, Anthropic) to build and run the audit harness, extract the receipts, and draft this preprint under the author's direction. Every claim rests on a re-runnable receipt: the pinned image, the reference solution, the mutation, and the grader's own reward. None rests on the model's say-so.

Funding

Conducted independently, with no external, institutional, or commercial funding. Compute and API costs were borne by the author.