SIDECAR: INDEPENDENT VERIFICATION FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents take many dependent steps, so one mistaken assumption can invalidate the work built on it. Checking it early is far cheaper than redoing that work. The natural remedy is to have the agent review its own work. Yet this review reads the same history that produced the mistake, so it tends to confirm the assumption rather than test it. To make verification reliable, we introduce SIDECAR, independent verification that keeps its own context. SIDECAR tests the agent's assumptions with executable checks and returns the evidence to the agent at decision points. Effective checks cannot be fully specified in advance, so SIDECAR improves its checks within the task using execution feedback. To prevent this improvement from bypassing its own constraints, SIDECAR fixes its observation and intervention interfaces, so that only the checks themselves are revised. Using the agent's own model for verification, SIDECAR improves task success for every backbone we test. On Terminal-Bench 2.0, it raises task success from 17.3% to 25.0%, a gain of 7.7 percentage points. Our code is available at this link.

1 INTRODUCTION

Large language model (LLM) agents are increasingly applied to long-horizon tasks that require many steps to complete (Kwa et al., 2025; Chen et al., 2026b; Jimenez et al., 2024; Deng et al., 2025; Zhou et al., 2024b; Xie et al., 2024; Xu et al., 2025; Merrill et al., 2026; Li et al., 2026b; Starace et al., 2025; Su et al., 2026). As the number of steps grows, errors accumulate: success rates fall with task length, and an early mistake often propagates to later steps (Kwa et al., 2025; Sinha et al., 2026; Laban et al., 2026; Wang et al., 2026c; Zhu et al., 2025; Rahman et al., 2026).

A mistaken assumption can become a premise for later actions, making recovery expensive. A natural remedy is to have the agent review its own work (Madaan et al., 2023; Shinn et al., 2023). Yet agents often fail to notice their own mistakes (Huang et al., 2024; Stechly et al., 2024; Rahman et al., 2026): the review reads the same history that produced the mistake, and models make more errors when their own earlier errors are already in context (Sinha et al., 2026; Laban et al., 2026).

For example, an agent may convert 1,000 records with a conversion script that silently skips 60 of them. The script's log reports "1,000 records processed," counting inputs rather than outputs, and the agent takes this as confirmation and moves on to the next step. Rereading the log only repeats this conclusion; comparing the actual input and output reveals the 60 missing records. This raises the question: *How can an agent's work be verified effectively during a long-horizon task?*

We address this question with SIDECAR, an independent verification process that runs beside the agent during the task. Because self-review reads the history that produced the mistake, SIDECAR keeps verification in a *separate context*: the verifier works from the task description and its own observations of the environment, and derives from the task requirements the claims to be verified. The verifier also needs evidence of its own, since reading the log in the example would lead it to the same conclusion as the agent. It obtains this evidence by running *executable checks* in the environment itself, and verifies each claim by the result of its check. SIDECAR runs these checks at *decision points*, where the agent is about to commit to a result, and returns the resulting evidence to the agent, so that a mistaken result can be corrected before it becomes a premise for later actions.

However, the checks themselves may be incorrect, and an incorrect check returns misleading evidence to the agent. A verifier cannot always write effective checks on the first attempt, but it can improve

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

them during the task using their execution feedback. Such improvement needs limits, since self-evolving agents have been shown to drift in unintended ways (Shao et al., 2025): a verifier that revises itself freely could also change the constraints it is meant to follow. This raises a second question: *How can the verifier improve its checks during the task while keeping its constraints fixed?*

SIDECAR performs *within-task self-improvement*: as its checks run during the task, the verifier revises them using their execution feedback and retains the revised checks for later decision points. To keep its constraints fixed, SIDECAR requires each revision to preserve the claim being verified, and runs every check in a fixed framework that controls what the verifier can observe, records each result, and decides which results may intervene. Self-improvement therefore changes how claims are verified, while what the verifier can observe and what its results authorize remain fixed.

We evaluate SIDECAR on TERMINAL-BENCH (Merrill et al., 2026) and SWE-bench Verified (OpenAI, 2024). It raises task success for all three backbone models, for example from 17.3% to 25.0% for gpt-oss-120b and from 77.1% to 80.7% for GPT-6-Astra on TERMINAL-BENCH 2.0, with the same model as its verifier. At the same decision points, SIDECAR outperforms self-reflection and a check-free rejection rule while making fewer rejections. Its checks also improve during the task, supplying evidence at later decision points while its constraints stay fixed.

Our contributions are:

- SIDECAR, an independent verification process that verifies an agent's claims in a separate context with executable checks and returns the evidence to the agent at decision points.
- Within-task self-improvement, in which the verifier revises its checks using execution feedback while a fixed framework keeps its constraints unchanged.
- An evaluation on TERMINAL-BENCH and SWE-bench Verified showing that SIDECAR raises task success for three backbone models, including the frontier agent GPT-6-Astra, which it raises from 79.8% to 84.5% on TERMINAL-BENCH 2.1, and, at the same decision points, outperforms both self-reflection and a check-free rejection rule.

2 RELATED WORK

Self-reflection and self-correction. Self-Refine and Reflexion let an agent critique and revise its own output, or carry verbal reflections into later attempts (Madaan et al., 2023; Shinn et al., 2023). Self-Debug has a model explain its code and repair it from execution results (Chen et al., 2024), and CRITIC checks outputs with external tools (Gou et al., 2024). In these methods, the model that made a decision also judges it, from the same context. Without an external signal, such self-correction often fails to improve reasoning and can even degrade it (Huang et al., 2024), which motivates pairing models with external verifiers (Kambhampati et al., 2024). Chain-of-Verification answers verification questions without seeing its draft, so that it does not repeat the draft's errors (Dhuliawala et al., 2024), but its answers still come from the model's own knowledge. SIDECAR separates verification from the agent's context and grounds it in executable checks on the environment.

Test-time verification. A common use of verification at test time is to select among finished attempts, using self-consistency, reward models, or generative verifiers (Wang et al., 2023; Lightman et al., 2024; Zhang et al., 2025a; Snell et al., 2024; Kwok et al., 2026). Such selection acts only after the runs end, multiplies the cost of each task, and repeats the same bias across samples drawn from one model. Closer to our setting, runtime guards and monitors judge an agent's actions while it runs, blocking unsafe steps, intercepting tool calls, or steering a drifting plan (Fang et al., 2025; Lee et al., 2026a; Zheng et al., 2026; Naik et al., 2026; Liu et al., 2026), and AI control lets a trusted monitor veto an untrusted policy (Greenblatt et al., 2024; Bhatt et al., 2025). These monitors typically read the agent's own context and return a natural-language verdict, yet model-written feedback is often the weak link in repair (Olausson et al., 2024; Dai et al., 2026; Lucassen & Kaufman, 2026). SIDECAR verifies during the run from its own context and returns environmental evidence to the agent at decision points.

Self-improving agents. Agents improve with experience by storing reflections for later trials (Shinn et al., 2023), executable skills (Wang et al., 2024), reusable workflows (Wang et al., 2025), or a persistent knowledge base that guides skill updates (Tang et al., 2026). Other work argues for optimizing the harness around the model (Gu, 2026); StateM turns selected postmortem findings

into persistent, executable preconditions (Qin et al., 2026), and a proof agent rewrites its workflow, prompts, and tools inside a small fixed, trusted runtime (Li et al., 2026a). These methods change the agent itself, through its context or harness, and most carry the improvement across trials or tasks. When the agent decides what enters its skill library, poisoned inputs can persist there as agent-written skills, even after the originals are removed (Ying et al., 2026; Wu et al., 2026), and self-evolution more broadly can deviate in unintended ways (Shao et al., 2025). Constraints held only in the agent's context can also be silently dropped during compaction (Chen, 2026; Wang et al., 2026d), whereas runtime enforcement checks agent actions against rules specified in advance (Wang et al., 2026b). SIDECAR instead improves within the current task, using the execution feedback of its own checks to revise those checks rather than the agent, and fixes how it observes and intervenes so that this improvement stays within its own constraints.

Generated tests and rubrics. Other work checks results with generated tests, executable verifiers, or rubrics. CodeT and UTGen generate unit tests to select or debug code (Chen et al., 2023; Prasad et al., 2025), SWT-Bench turns user issues into tests that filter proposed fixes (Mündler et al., 2024), and other methods derive postconditions, executable verifiers, or rubrics from a task's natural-language description (Endres et al., 2024; Pezeshkpour & Hruschka, 2026; Wan et al., 2026; Zhao et al., 2026a). These checks evaluate a produced result, such as a program, a patch, or an answer, whereas SIDECAR tests the agent's intermediate assumptions during a long-horizon task and reruns its checks on the current state as the run proceeds. Appendix A extends this discussion with a broader comparison.

3 PROBLEM SETUP AND PRELIMINARY STUDY

3.1 PROBLEM SETUP

An agent performs a task by acting in an environment over many steps. The public task description states its **requirements**, conditions the agent's work must satisfy, such as "convert all 1,000 records." A **mistaken assumption** is a false belief, formed during the task, that a requirement is met. For example, a conversion script silently skips 60 records but logs "1,000 records processed," so the agent assumes all records are converted. It carries this belief to its next **decision point**, a step where it is about to commit to a result, such as handing the records to the next stage or declaring completion.

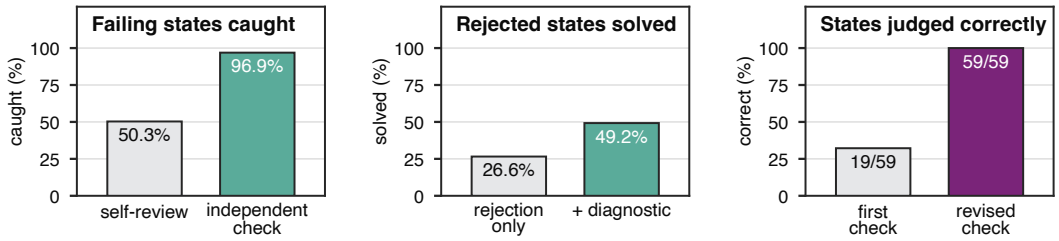
Verification aims to expose mistaken assumptions at decision points, returning evidence while the agent can still act on it. The verifier works only from the public task description and the environment, in a context separate from the agent's. It derives **claims**, statements about the environment implied by the requirements, such as "the output contains 1,000 records," and tests each with a **check**: executable code that runs in the environment and returns its raw output as evidence. Before the handoff, a record-counting check reports "940 of 1,000 records found." Task outcomes are labeled for evaluation only, by the hidden official tests or a public-requirement predicate. The objective is to raise task success while limiting verification cost and the harm of unnecessary intervention to correct work.

3.2 PRELIMINARY STUDY: FROM SELF-REVIEW TO INDEPENDENT VERIFICATION

We run four preliminary experiments on TERMINAL-BENCH 2.0 (Merrill et al., 2026), with results in Figure 1. They ask, in turn, whether the agent's own review catches its failures, whether an independent check detects them, whether returning the check's output helps the agent recover, and whether checks improve from their execution feedback. The first experiment runs the agent alone; the other three use environment states captured at the decision point where the agent declares completion, each labeled as failing or passing by a predicate over the task's public requirements.

The agent's own review accepts most of its failing work. TERMINUS-2 already gives the agent a chance to catch its mistakes: before a task is marked complete, it asks the agent to confirm, and the agent reviews its work in its own context. We measure how often this review rejects completion on failing work, running gpt-oss-120b with TERMINUS-2 over three seeds. The agent solves 17.3% of tasks, and 87% of its failing runs end with the agent confirming completion. In 10 of 21 annotated failing runs, the output revealing the failure was already in the agent's context when it confirmed.

Executable checks in a separate context detect nearly all failing states. Because the agent's own review lets these failures through, we test whether a check written outside the agent's context catches



(a) Detecting failing states. (b) Recovering after a rejection. (c) Revising a check.

Figure 1: Preliminary experiments on TERMINAL-BENCH 2.0.

them. For each task, gpt-oss-120b writes one executable check from the public task description, in a context separate from the agent's; the check is frozen and then run in a copy of each captured environment. We compare it with the agent's own review on the same failing states of gpt-oss-120b, where the agent is explicitly asked to check its work. As shown in Figure 1a, self-review catches 50.3% of these failures, while the independent check catches 96.9% of them.

Returning the check's evidence at a decision point helps the agent recover. Detecting a failure helps only if the agent can act on it, so we test whether the check's evidence changes what the agent does next. For 128 states that a check rejects, we reject the agent's completion and let it continue from the same state, twice with the same seeds. In one run, the rejection only says that the task is not complete; in the other, it also includes the unedited output of one diagnostic that fixed code selects for the failing requirement. Because both runs are rejected at the same state and continue under the same budget, any difference between them comes from the evidence. A predicate over the public requirements judges each final state. As shown in Figure 1b, after the plain rejection the agent solves 26.6% of these states; with the diagnostic, it solves 49.2%, nearly twice as many states.

Execution feedback makes checks more accurate. The previous experiment shows that a check's evidence helps the agent correct its work. This one tests whether a check's own execution feedback helps the verifier correct the check. A check is written from the task description alone, so, like the agent's work, it can rest on a mistaken assumption, and running it exposes that assumption. On task `regex-log`, for example, a check probes the agent's regular expression by passing a compiled pattern together with a flags argument to `re.findall`, which Python rejects. Running it returns no verdict and the error "cannot process flags argument with a compiled pattern." Given this error together with the claim and its own source, the verifier passes the pattern as a string instead, and the revised check reaches a verdict that agrees with the official test. In a controlled measurement on `headless`, shown in Figure 1c, revision from execution feedback on labeled development states raises correctly judged held-out states from 19 to 59 of 59.

Takeaway: The agent's own review lets most failing work through. Verification works better when executable checks run in a separate context, return their evidence at decision points, and improve from execution feedback.

4 SIDECAR: INDEPENDENT VERIFICATION

Our preliminary study demonstrates that executable checks operating in a separate context detect failures missed by self-review, that returning evidence at decision points aids recovery, and that execution feedback improves check accuracy. The first two findings motivate the design of verification at decision points, while the third informs our approach to within-task learning. We begin with an overview of SIDECAR before detailing two core mechanisms: how independent verification evaluates an agent's work during long-horizon tasks, and how the verifier iteratively improves its checks without compromising system constraints.

4.1 OVERVIEW

SIDECAR runs beside the agent throughout a task to verify its intermediate and final outputs, as illustrated in Figure 2. It comprises two components with separate roles. The **verifier**, instanti-

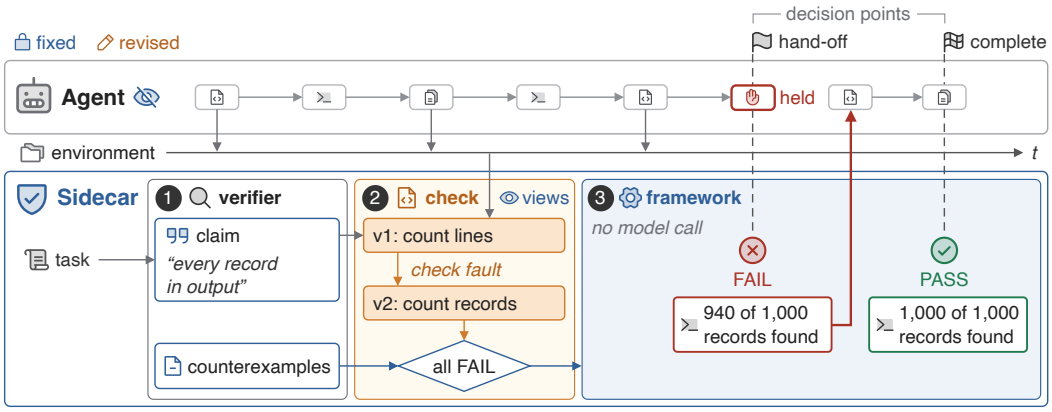


Figure 2: Overview of SIDECAR. Dashed lines mark decision points. At the hand-off, a failing check holds the agent's step and returns its output to the agent; at completion, every check passes. Blue parts stay fixed throughout the task, while orange parts, the checks, are revised from their execution feedback.

ated with the agent's own model, decides what to check. It reads the task description, turns its requirements into claims, and writes an executable check for each claim. It works in its own context and never sees the agent's reasoning. The **framework**, a static codebase that the verifier cannot modify, determines *how* checks are executed. At each decision point, the framework executes the generated checks against the current environment state. If a check fails, the framework halts the agent's execution and returns the check's output as corrective evidence. As the task progresses, the verifier can revise faulty checks based on execution feedback. To guard against weakening a check, the framework accepts a revision only if the revised check still rejects the claim's predefined counterexamples, environment states constructed to violate the claim. Only the checks change during the task; the framework, the claims, and their counterexamples remain fixed.

4.2 VERIFICATION AT DECISION POINTS

SIDECAR verifies the agent's work independently of the agent's own reasoning and returns feedback before the agent commits to a result. It does so through three design choices: claims are derived from the task description, evidence is generated by executable checks against the environment, and feedback is returned at decision points.

Claims from requirements. A claim is a testable statement about the environment that must hold for a task requirement to be satisfied. For example, a requirement to convert every input record yields the claim that every record appears in the output, as illustrated in Figure 2. To keep the agent's reasoning from influencing what is checked, the verifier writes its claims before the agent takes its first action, using only the task description and a snapshot of the initial environment. SIDECAR therefore needs no hidden tests or reference solutions. Each claim quotes the requirement it tests, making every later judgment traceable to the task description. Along with each claim, the verifier writes an initial executable check and a few counterexamples, which are used to test later revisions of the check.

Evidence from executable checks. A check is a short program designed to test a single claim by directly interrogating the environment. This circumvents the agent's internal logs, which may report success despite underlying failures. The framework exposes the environment to the checks through three static views: a file listing of the environment snapshot, specific file contents, and the output of a command that the check executes, such as compiling the agent's code or running it on new inputs. Because these views extract data directly from the environment state, every observation is grounded in the environment. The verifier decides what each check inspects, while the framework fixes which views are available, so the verifier's access to the environment stays the same even as its checks are revised.

Each check is executed inside a disposable sandbox of the current environment state, allowing it to test the agent's artifacts without altering the agent's workspace. The raw output printed by the check, such as "940 of 1,000 records found" in Figure 2, serves as evidence for the claim.

Feedback at decision points. A decision point is a critical juncture where the agent prepares to commit to a result, such as passing data to a downstream pipeline or declaring task completion; Figure 2 shows one of each. Once a commitment is made, subsequent steps treat the result as a premise, so an error found afterward requires redoing the work built on it. The framework intervenes at these points, where errors can still be corrected efficiently.

The framework keeps notes for each claim, recording the quoted requirement, its counterexamples, every version of its check, and every result with the check's output. At each decision point, it re-evaluates all stored checks associated with the pending commitment against the current environment state. Re-evaluating these checks requires no additional model call and tests for the recurrence of previously detected failures. Each execution yields one of three results: PASS, FAIL, or UNKNOWN when the check cannot reach a conclusion. Only a FAIL triggers a rejection. The framework returns exactly what the check printed, and the agent decides how to proceed. The framework provides this raw output without generating a supplementary model explanation, because such an explanation can add an incorrect interpretation that the agent may follow.

4.3 Learning from Execution Within a Task

Because checks are written zero-shot prior to execution, the initial versions may contain syntax errors or fail to accurately test the intended claim. Therefore, optimal check logic cannot be fully specified in advance. SIDECAR leverages execution feedback to iteratively improve check accuracy during the task, while employing a fixed execution framework to prevent the verifier from relaxing its own constraints.

Improving checks. Execution feedback exposes two primary failure modes in check logic. First, a check may fail to execute (returning UNKNOWN), in which case the error message helps identify the fault. Second, a check may execute successfully but measure the wrong metric. For example, in Figure 2, the initial check counts lines instead of records; the check's output reveals this discrepancy, prompting the verifier to revise the logic to count records. During a revision cycle, the verifier is provided with the claim, the faulty check code, and the execution feedback, and is prompted to generate an updated version of the check. Because every execution yields new feedback, this revision process continues dynamically throughout the task.

Keeping Constraints Fixed. An unrestricted self-improvement loop poses a significant risk: the verifier could "improve" a failing check by simply deleting its assertions, making it trivially easy to pass. SIDECAR prevents this reward hacking at two levels. First, the verifier's write access is strictly limited to the check scripts themselves. The observation interfaces, sandbox environment, result routing, and intervention triggers are hardcoded into the static framework. Consequently, a revision can change *how* a claim is tested, but cannot alter what the check is permitted to observe or how its verdict is utilized. Second, the framework enforces rigid counterexample testing. Alongside the verifier-generated counterexamples, the framework automatically injects two standardized counterexamples for every file referenced in a claim: deleting the file and emptying it. Before a new or revised check is deployed to the primary agent, the framework tests it against the current environment state and all associated counterexamples. The check is only accepted if it returns FAIL for every counterexample. This validation requires no labeled data, as the counterexamples are explicitly constructed to violate the claim. By freezing the claims, counterexamples, and framework architecture, SIDECAR ensures that within-task adaptation optimizes only the accuracy of the checks, not the strictness of the constraints.

5 Experiments

Evaluation overview. We frame the evaluation around four questions about SIDECAR. *(i) Task success.* Does SIDECAR improve task success across backbones and benchmarks? *(ii) Source of the gain.* Where does the gain of SIDECAR come from? *(iii) Check improvement.* Do the checks of SIDECAR improve during a task? *(iv) Cost.* What does verification cost? The first two questions evaluate independent verification at decision points, the third evaluates within-task self-improvement, and the fourth measures what both cost. The preliminary study tested detection, recovery and check revision separately on captured states; the runs below test the whole system end to end.

Benchmarks. TERMINAL-BENCH 2.0 (Merrill et al., 2026) contains 89 terminal tasks, each run in its own container and scored by hidden official tests; a task passes when all of its official tests pass. Five offensive-security tasks are excluded from all reported results because GPT-6-Astra declines them under its safety policy, leaving 84 tasks for every backbone; Appendix C.1 names the excluded tasks. TERMINAL-BENCH 2.1 revises part of these tasks and hosts the current public leaderboard; we use it on the same 84 tasks to compare a frontier agent with the leaderboard entries. SWE-bench Verified (Jimenez et al., 2024; OpenAI, 2024) contains 500 human-validated issues from open-source Python repositories. The agent receives the issue text and edits the repository, and a task passes when all of its fail-to-pass and pass-to-pass tests pass after the hidden test patch is applied. Both benchmarks run through HARBOR 0.18.0 (Laude Institute, 2025), which adds a task's hidden tests to its container only after the agent finishes, so neither the agent nor the verifier of SIDECAR can read them.

Models and systems. We evaluate three backbones: gpt-oss-20b and gpt-oss-120b (OpenAI, 2025), and GPT-6-Astra. Every model call in a run uses the agent's backbone, so comparisons within a backbone hold model capability fixed. All systems run the TERMINUS-2 scaffold with the same agent prompts, turn limit, agent sampling settings, container resources and per-task time limits. The *Baseline* is native TERMINUS-2, which asks the agent to confirm completion before a task ends, so the agent reviews its own work in its own context. SIDECAR acts at the agent's confirmed completions, which are the decision points of these runs. Its verifier works in a separate context: before the agent starts, it writes claims and checks from the task description, and it revises the checks from their execution feedback during the task. At each confirmed completion, the framework runs the checks in a copy of the environment, rejects the completion if a check fails, and returns the check's output to the agent, which then continues the task. Verification counts against the task's unchanged time limit. Appendix C collects the success rates in Table 7 and provides the benchmark scope, agent settings, completion controls, verification procedure and metric definitions. Unless a figure caption states otherwise, success rate is the fraction of passing trials among those with an official reward in the reported comparison.

6 RESULTS

The evaluation asks four questions, about task success, the source of the gain, check improvement and cost, and we answer them in this order. Section 6.1 reports task success across backbones and benchmarks, Section 6.2 locates the source of the gain by comparing SIDECAR with two systems that also reject completions at the same decision points, Section 6.3 follows how the checks improve during a task, and Section 6.4 measures the calls, tokens and time that verification adds.

6.1 MAIN RESULTS

SIDECAR improves a frontier agent. We first ask whether SIDECAR still helps when both the model and the harness are already strong. On TERMINAL-BENCH 2.1, GPT-6-Astra with TERMINUS-2 solves 79.8% of our 84 tasks on its own, above the TERMINUS-2 entries of the leaderboard recomputed on the same tasks in Figure 3, and SIDECAR raises its success rate to 84.5%. It does so with a single rejection over all 84 tasks: a strong agent only rarely confirms completion on work that still fails a check, so verification seldom needs to interrupt the agent at all.

The gain holds across backbones. We next ask whether the gain depends on the backbone. As shown in Figure 4, on TERMINAL-BENCH 2.0 SIDECAR raises the success rate of gpt-oss-20b from 5.6% to 9.7%, of gpt-oss-120b from 17.3% to 25.0%, and of GPT-6-Astra from 77.1% to 80.7%. Over the three backbones together, it wins 30 paired trials and loses 10, which is significant with $p = 0.002$ under an exact sign test. The size of the gain likely depends on two things: how many failures are left to catch, and whether the agent can fix a failure once a check exposes it. gpt-oss-20b fails most tasks but often cannot fix what a check reports, GPT-6-Astra has few failures left, and gpt-oss-120b, with both room and ability, gains the most. Because each verifier uses the same model as its agent, none of these gains relies on a stronger model checking the work of a weaker one.

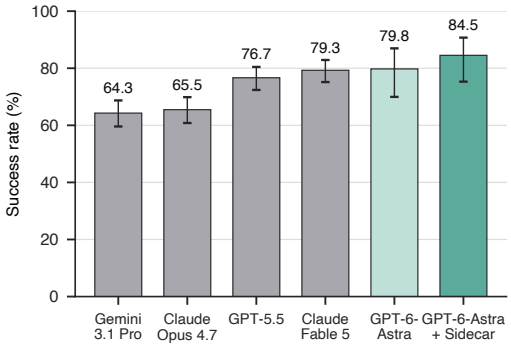


Figure 3: Success rate on TERMINAL-BENCH 2.1 with TERMINUS-2, counting unscored trials as failures: leaderboard entries recomputed on our tasks in gray, GPT-6-Astra without and with SIDECAR in teal.

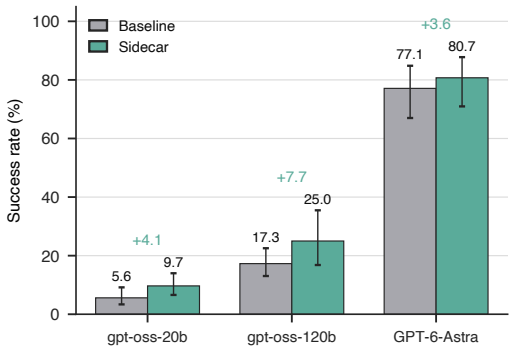


Figure 4: Success rate of the Baseline and SIDECAR with three backbones on TERMINAL-BENCH 2.0. Numbers give the gain in percentage points.

SIDECAR on SWE-bench Verified. We then ask whether the gain carries over to a different kind of task. SWE-bench Verified asks the agent to resolve real issues in open-source Python repositories, and we run SIDECAR on it with the same configuration as on TERMINAL-BENCH 2.0. As shown in Figure 5, SIDECAR raises the success rate of gpt-oss-20b from 4.2% to 6.2% and of gpt-oss-120b from 11.6% to 15.1%, and stays on par with GPT-6-Astra, at 83.8% for both systems. Over the three backbones, it wins 78 paired issues and loses 51, which is significant with $p = 0.022$ under an exact sign test. The same two factors fit these results: GPT-6-Astra has few failures left, and both gpt-oss models gain less than on TERMINAL-BENCH 2.0, where they also solve more of the tasks on their own.

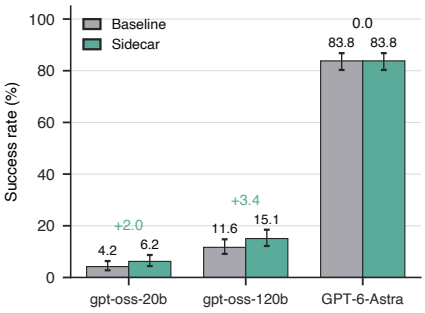


Figure 5: Success rate of the Baseline and SIDECAR on SWE-bench Verified. Numbers on top give the gain in percentage points.

6.2 SOURCE OF THE GAIN

Any system that rejects a confirmed completion gives the agent more turns, so the gain of SIDECAR could come from these turns alone, from a review of the agent's work, or from independent verification. We separate these sources with two systems that reject completions at the same decision points, run with gpt-oss-120b on TERMINAL-BENCH 2.0. The *Rejection rule* gives only more turns: it makes no model call, runs no check and returns a generic message. *Self-reflection* adds a review in the agent's own context: the agent's model rereads the full transcript and returns its reflection. SIDECAR replaces this review with independent verification by executable checks. Figure 6 reports the success rate of each system and how many confirmed completions it rejects in each trial.

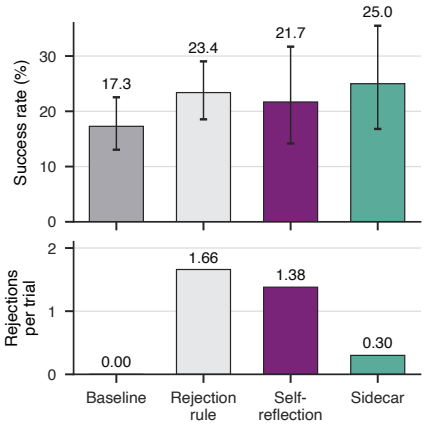


Figure 6: The four systems at the same decision points, all with gpt-oss-120b on TERMINAL-BENCH 2.0: success rate at the top and completions rejected per trial below.

Verification reaches the gain with fewer rejections. SIDECAR raises success from 17.3% to 25.0%, against 23.4% for the Rejection rule and 21.7% for Self-reflection. Part of this gain is available to any system that rejects at a confirmed completion, since most failing runs end with the agent confirming completion on work that fails. What distinguishes the systems is which completions they reject. The rule rejects without looking at the work, correct or not, 1.66 times per trial; Self-reflection reviews the work in the agent's own context, rejects 1.38 times per trial and does not exceed the rule; SIDECAR rejects only when one of its checks fails, which happens only 0.30 times per trial.

Each rejection tells the agent what is wrong. The three systems also differ in what a rejection tells the agent. The rule returns only a generic message, Self-reflection returns the model's own reflection, and SIDECAR returns the unedited output of the check that failed. On captured failing states, where the state, the turns and the seeds are held fixed, the agent solves 26.6% of the states after a plain rejection and 49.2% after a rejection that also carries a diagnostic's output, as shown in Figure 1b. Since SIDECAR rejects only when a check fails, it also spends fewer rejections per trial than either of the two controls, as shown in the lower panel of Figure 6, and so interrupts less of the work that is already correct. The gain therefore arises at the decision point, and independent verification directs each rejection at work that fails and tells the agent what is wrong.

6.3 How Checks Improve Within a Task

Execution feedback repairs faulty checks. A check helps only if it reaches a verdict when the agent commits, but a check written before the agent starts often cannot, because the verifier has not yet run any code in the environment. We follow the checks of SIDECAR in the same gpt-oss-120b run on TERMINAL-BENCH 2.0. The verifier first runs each check on the initial environment, before the agent acts, and revises it whenever its execution exposes a fault in the check itself. As shown in Figure 7, this lowers the share of checks that cannot reach a verdict from 41.5% to 22.3%, and most repaired checks return FAIL, as they should before the work is done. At decision points, the retained checks rerun on the agent's work, and 88.9% of these executions return PASS or FAIL. The repaired checks supply 8 of the 24 rejections, and each keeps the claim of the check it replaces, so revision changes only how a requirement is tested.

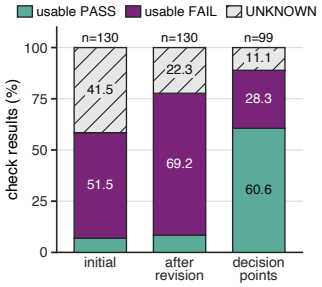


Figure 7: Check results with gpt-oss-120b on TERMINAL-BENCH 2.0. The first two bars count checks; the last counts executions at decision points.

6.4 What Verification Costs

The verifier itself is cheap. Verification adds model calls, tokens and time to each run, which we measure in the same gpt-oss-120b runs on TERMINAL-BENCH 2.0 and SWE-bench Verified. The verifier makes a mean of 2.4 and 2.0 model calls per trial, 7.7% and 6.9% of the agent's calls, and accounts for 1.2% of the tokens of a SIDECAR run on SWE-bench Verified, because it reads the public task, fixed views of the environment and its own checks rather than the agent's transcript.

The main cost is time taken from the agent. Verification runs within the agent's unchanged time limit and takes a median of 16.9% and 15.9% of a trial's wall-clock time on the two benchmarks. With less time for its own work, the agent reaches the time limit more often, in 15.0% of trials instead of 9.9% on TERMINAL-BENCH 2.0 and in 4.2% instead of 2.6% on SWE-bench Verified, mostly in trials where SIDECAR never rejects a single completion during the task.

7 Conclusion

We presented SIDECAR, an independent verification process for long-horizon agents. It tests claims derived from the task requirements with executable checks in a separate context, returns the evidence to the agent at decision points, and improves its checks within the task while its constraints stay fixed. With the agent's own model as its verifier, SIDECAR raises task success on both benchmarks, including for a frontier agent: on TERMINAL-BENCH 2.1, it raises GPT-6-Astra from 79.8% to 84.5%. At the same decision points, it outperforms self-reflection and a check-free rejection rule while rejecting far fewer completions, because it rejects only when a check fails and returns that check's unedited output. Within a task, execution feedback repairs faulty checks, lowering the share that cannot reach a verdict from 41.5% to 22.3%, while the claims and the framework stay fixed. Because the verifier reads the public task and its own checks rather than the agent's transcript, it adds few model calls; its main cost is the time it takes from the agent. Keeping verification apart from the agent's reasoning and grounding it in execution thus lets the agent's own model verify its work during a long-horizon task.

Ethics Statement

This work involves no human subjects or personal data. Experiments use public benchmark tasks in isolated containers. Executable checks run in disposable copies of task state, so their actions do not modify the agent's workspace. Official tests score completed trials rather than supply feedback to the agent or verifier. The benchmark scope is described in Appendix C.1. An intervention can also disrupt correct work; the preliminary experiments report rescue and disruption together and examine a concrete harm mechanism in Appendix B.4.1.

Reproducibility Statement

Appendix B specifies the captured-state experiments, their public-requirement predicates, continuation messages, and check-generation protocol. It identifies the development and sample-selection conditions needed to interpret those results. Appendix C describes the benchmark scope, agent settings, completion controls, verification procedure and reported metrics. Table 7 collects the success rates plotted in the main text. Code is linked in the abstract; experiment configurations and run artifacts will accompany the camera-ready version.

The Use of Large Language Models

Generative AI tools were used to author and revise experiment code, to analyze experiment results, to draft and edit the paper, and to search and summarize related work. The authors reviewed all AI-assisted work. Language models also form part of the systems studied: the SIDECAR verifier writes and revises checks using the agent's backbone, whereas the check generator and note writer of the preliminary experiments use gpt-oss-120b, as specified in Appendix B.6. The explanation comparison is reported in Appendix B.4.2. In SIDECAR, feedback is collected from executing model-written checks without an additional model-generated explanation. We take responsibility for the final text, code, claims and artifacts, including material produced with generative AI.

References

Laksh Advani. From confident closing to silent failure: Characterizing false success in LLM agents. *arXiv preprint arXiv:2606.09863*, 2026.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

Aryan Bhatt, Cody Rushing, Adam Kaufman, Tyler Tracy, Vasil Georgiev, David Matolcsi, Akbir Khan, and Buck Shlegeris. Ctrl-z: Controlling AI agents via resampling. *arXiv preprint arXiv:2504.10374*, 2025.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023.

Jianlyu Chen, Yuyang Hu, et al. Repo-to-skill: Distilling GitHub repositories into AI4AI skills. *arXiv preprint arXiv:2609.02749*, 2026a.

Mingguang Chen, Licheng Wang, and Bo Qu. The horizon gap: Planning, memory, execution, training, and evaluation for long-horizon LLM agents. *arXiv preprint arXiv:2608.06663*, 2026b.

Shiyang Chen. Governance decay: How context compaction silently erases safety constraints in long-horizon LLM agents. *arXiv preprint arXiv:2606.22528*, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. FeedbackEval: A benchmark for evaluating large language models in feedback-driven code repair tasks. *arXiv preprint arXiv:2504.06939*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941v2*, 2025. URL https://arxiv.org/abs/2509.16941v2.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. In *Findings of the Association for Computational Linguistics (ACL)*, 2024.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering (FSE)*, 2024.

Haishuo Fang, Xiaodan Zhu, and Iryna Gurevych. Preemptive detection and correction of misaligned actions in LLM agents. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025. doi: 10.18653/v1/2025.emnlp-main.12.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations (ICLR)*, 2024.

Ryan Greenblatt, Buck Shlegeris, Kshitij Sachan, and Fabien Roger. AI control: Improving safety despite intentional subversion. In *International Conference on Machine Learning (ICML)*, 2024.

Shangding Gu. From model scaling to system scaling: Scaling the harness in agentic AI. *arXiv preprint arXiv:2605.26112*, 2026.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, Huan Sun, and Yu Su. Is your LLM secretly a world model of the internet? Model-based planning for web agents. *Transactions on Machine Learning Research*, 2025.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

Mehmet Iscan. Falsification, not exposure: An internally preregistered placebo-controlled decomposition of self-repair feedback in frozen small code models. *arXiv preprint arXiv:2606.31511*, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. LLMs can't plan, but can help planning in LLM-modulo frameworks. In *International Conference on Machine Learning (ICML)*, 2024.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*, 2024.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Chris Painter, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long software tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2503.14499.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, et al. LLM-as-a-verifier: A general-purpose verification framework. *arXiv preprint arXiv:2607.05391*, 2026.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2505.06120.

Laude Institute. Harbor: A framework for building, evaluating and training agents in containerized environments. https://github.com/laude-institute/harbor, 2025.

Jaewoo Lee, Zaid Khan, Archiki Prasad, Justin Chih-Yao Chen, Supriyo Chakraborty, Kartik Balasubramaniam, Sambit Sahu, Elias Stengel-Eskin, Hyunji Lee, and Mohit Bansal. A history-aware visually grounded critic for computer use agents. *arXiv preprint arXiv:2606.11078*, 2026a.

Nicholas Lee, Lutfi Eren Erdogan, Chris Joseph John, Surya Krishnapillai, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. Agentic test-time scaling for WebAgents. *arXiv preprint arXiv:2602.12276*, 2026b.

Yuqing Li, Zeguan Wu, Yu Gan, and Junyu Liu. Self-modifying Lean proof agents with verifier-grounded benchmark coevolution. *arXiv preprint arXiv:2607.17352*, 2026a.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, et al. Long-horizon-terminal-bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026b.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024.

Shuyang Liu, Saman Dehghan, Ji Young Kim, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. Online monitoring and corrective steering of programming agents. *arXiv preprint arXiv:2608.06701*, 2026.

James Lucassen and Adam Kaufman. Retrying vs resampling in AI control. *arXiv preprint arXiv:2605.26047*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM critics help catch LLM bugs. *arXiv preprint arXiv:2407.00215*, 2024.

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

Evan Miller. Adding error bars to evals: A statistical approach to language model evaluations. *arXiv preprint arXiv:2411.00640*, 2024.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Vishal Ishwar Naik, Chenyu Xu, Donna Dong, Hussein Hassan, Abhishek Pradhan, Ofer Mendelevitch, Tallat Shafat, and Humayun Irshad. GuardianAgentBench: Where agents fail and how to guard them. *arXiv preprint arXiv:2607.20982*, 2026.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*, 2024.

OpenAI. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, 2024.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. arXiv preprint arXiv:2508.10925, 2025.

Pouya Pezeshkpour and Estevam Hruschka. AutoPyVerifier: Learning compact executable verifiers for large language model outputs. *arXiv preprint arXiv:2604.22937*, 2026.

Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging. In *Conference on Language Modeling (COLM)*, 2025.

Ziheng Qin, Yaxin Lu, Zhangyang Atlas Wang, and Kai Wang. StateM: Reaching 95.3% raw accuracy, or a $15 frontier run, on terminal-bench 2.1 via harness scaling. *arXiv preprint arXiv:2608.15089*, 2026.

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

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Shuai Shao, Qihan Ren, Chen Qian, et al. Your agent may misevolve: Emergent risks in self-evolving LLM agents. *arXiv preprint arXiv:2509.26354*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in LLMs. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2509.09677.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's ability to replicate AI research. In *International Conference on Machine Learning (ICML)*, 2025. URL https://arxiv.org/abs/2504.01848.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115*, 2024.

Liangcai Su, Zhaopeng Feng, Zhuo Chen, Zhen Zhang, et al. FrontierChallenge: Evaluating scientific workflow completion. *arXiv preprint arXiv:2608.24979*, 2026.

Zhensu Sun, Chengran Yang, Yunbo Lyu, Jieke Shi, and David Lo. Second thought: Reasoning in parallel as LLM agents act and observe. *arXiv preprint arXiv:2608.13667*, 2026.

Liyan Tang, Cyrus Rashtchian, Chun-Sung Ferng, Andrew Tomkins, Da-Cheng Juan, and Tu Vu. WikiSkill: Compiling agent experience into persistent knowledge for skill evolution. *arXiv preprint arXiv:2608.27454*, 2026.

Yuxuan Wan, Tianqing Fang, Zaitang Li, Yintong Huo, Wenxuan Wang, Haitao Mi, Dong Yu, and Michael R. Lyu. Inference-time scaling of verification: Self-evolving deep research agents via test-time rubric-guided verification. *arXiv preprint arXiv:2601.15808*, 2026.

Binghai Wang, Chenlong Zhang, Dayiheng Liu, Jiajun Zhang, Jiawei Chen, Mingze Li, Mouxiang Chen, Rongyao Fang, Siyuan Zhang, Xuwu Wang, Yuheng Jing, Zeyao Ma, and Zeyu Cui. The verification horizon: No silver bullet for coding agent rewards. *arXiv preprint arXiv:2606.26300*, 2026a.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026b.

Xinyu Jessica Wang, Haoyue Bai, Yiyou Sun, Haorui Wang, Shuibai Zhang, Wenjie Hu, Mya Schroder, Bilge Mutlu, Dawn Song, and Robert D. Nowak. The long-horizon task mirage? Diagnosing where and why agentic systems break. *arXiv preprint arXiv:2604.11978*, 2026c.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhiqi Wang, Yichi Zhang, Dongwon Lee, and Yuchen Yang. Lost in compaction: Evaluating side-constraint loss under context compaction. *arXiv preprint arXiv:2608.11242*, 2026d.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning (ICML)*, 2025.

Xiaodong Wu, Yu Shi, Qi Li, Zhimin Zhao, Xiangman Li, Bram Adams, Ahmed E. Hassan, and Jianbing Ni. EvoMal: Self-poisoning in self-evolving coding agents. *arXiv preprint arXiv:2608.25776*, 2026.

Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024. doi: 10.1145/3650212.3680323.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2404.07972.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2412.14161.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Zonghao Ying, Xiangfan Wu, Huiyu Wu, Xing Zheng, Huangsheng Cheng, Xiaorong Shi, and Jing Guo. SkillJack: Persistent skill backdoors in self-evolving agents. *arXiv preprint arXiv:2608.03509*, 2026.

Xiao Yu, Baolin Peng, Vineeth Vajipey, Hao Cheng, Michel Galley, Jianfeng Gao, and Zhou Yu. ExACT: Teaching AI agents to explore with reflective-MCTS and exploratory learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations (ICLR)*, 2025a.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *International Conference on Machine Learning (ICML)*, 2025b.

James Xu Zhao, Hui Chen, Bryan Hooi, and See-Kiong Ng. FineVerify: Scaling test-time compute with fine-grained self-verification for agentic search. *arXiv preprint arXiv:2606.00660*, 2026a.

Xiangxin Zhao, Han Li, Shuaiting Li, Tianyi Zhao, Earl T. Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of CLI coding agent trajectories. *arXiv preprint arXiv:2607.09510*, 2026b.

Zhijie Zheng, Yu Li, Chen Qian, Yuqian Fu, Yanwei Fu, Lu Sheng, Jing Shao, and Dongrui Liu. StepGuard: Learning step-level guardrails with scalable supervision and safety-utility balancing. *arXiv preprint arXiv:2608.24777*, 2026.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning (ICML)*, 2024a.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024b.

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

SUPPLEMENTARY MATERIALS – TABLE OF CONTENTS

A Extended Related Work **17**

A.1 Long-Horizon Agents and Their Failures 17

A.2 Self-Reflection and Self-Correction 17

A.3 Test-Time Verification 18

A.4 Self-Improving Agents 19

A.5 Generated Tests and Rubrics 19

B Preliminary Experiments **19**

B.1 Tasks, Captured States and the Predicate Φ 19

B.2 Experiment 1: The Agent's Own Review 20

B.3 Experiment 2: Failure Detection 20

B.4 Experiment 3: Recovery after Rejection 21

B.4.1 Plain Rejection versus Diagnostic Feedback 21

B.4.2 Raw Output versus a Model-Written Explanation 21

B.5 Experiment 4: Check Revision from Execution Feedback 22

B.6 Check Generator and Note Writer 22

C Experimental Details **23**

C.1 Benchmarks and Task Scope 23

C.2 Agent Settings, Resources and Serving 23

C.3 Decision Points, Baseline and Controls 24

C.4 Configuration of SIDECAR 25

C.5 Scoring, Metrics and Statistics 26

C.6 Complete Results 26

C.7 Trial Outcomes 26

D Case Studies **28**

D.1 Verification and Recovery 28

D.2 Check Revision from Execution Feedback 31

Table 1: **Experiments in this paper.** *Question:* what the experiment tests. *Main result:* its headline number. *Section:* where it is reported, the main text first and the appendix second.

Experiment	Question	Main result	Section
Preliminary study, Section 3.2			
Exp. 1: own review	Does the agent's own review reject completion on failing work?	87% of failing runs end with the agent confirming completion.	3.2, B.2
Exp. 2: independent check	Does a separate check catch these failures?	It catches 96.9% of failing states, against 50.3% for self-review.	3.2, B.3
Exp. 3: returned diagnostic	Does the check's evidence help recovery?	On 128 rejected states, the agent solves 26.6% after a plain rejection and 49.2% with the diagnostic.	3.2, B.4
Exp. 4: check revision	Does execution feedback improve a check?	Correctly judged held-out states rise from 19 to 59 of 59.	3.2, B.5
Results, Section 6			
Task success	Does SIDECAR raise task success?	On TERMINAL-BENCH 2.0, gains of 3.6 to 7.7 pp; on SWE-bench Verified, 0 to 3.4 pp.	6.1, C.6
Frontier agent	Does verification help a strong agent?	On TERMINAL-BENCH 2.1, SIDECAR raises GPT-6-Astra from 79.8% to 84.5%.	6.1, C.6
Self-reflection, Rejection rule	Where does the gain of SIDECAR come from?	21.7% and 23.4%, against 25.0% for SIDECAR and 17.3% for the Baseline.	6.2, C.3
Checks during runs	Do checks improve within a task?	Checks returning UNKNOWN fall from 41.5% to 22.3%.	6.3, C.5
Cost	What does verification cost?	On TERMINAL-BENCH 2.0 and SWE-bench Verified, verifier calls are 7.7% and 6.9% of the agent's calls; verification takes a median 16.9% and 15.9% of trial time.	6.4, C.5
Appendix analyses			
Raw output or explanation	What should a rejection carry?	On 32 rejected states, the agent solves 21 with the raw diagnostic and 10 with a model-written explanation.	B.4.2

A EXTENDED RELATED WORK

This appendix expands Section 2, beginning with background on long-horizon failures.

A.1 LONG-HORIZON AGENTS AND THEIR FAILURES

In long-horizon tasks, an early mistake can propagate through the steps built on it, as discussed in Section 1. One visible symptom is a failing run that ends as if it had succeeded: on FrontierChallenge, 75.5% of non-passing Claude Code runs end with a message announcing completion (Su et al., 2026), and Advani (2026) characterizes the same false-success pattern in tool-use and coding agents. In our preliminary study, 87% of failing runs end with the agent confirming completion, as reported in Section 3.2. Zhao et al. (2026b) study failure in CLI coding agent trajectories as a process; failure taxonomies and failure attribution classify failures and locate the responsible step (Cemri et al., 2025; Zhang et al., 2025b); and pass^k (Yao et al., 2024) measures whether an agent succeeds consistently across k repeated trials.

A.2 SELF-REFLECTION AND SELF-CORRECTION

Besides the critique-and-revise methods of Section 2, confidence-triggered sampling and parallel self-reasoning spend more reasoning where the model is uncertain (Lee et al., 2026b; Sun et al., 2026). Their signal comes from the agent's own model, and a confidence trigger does not fire when the agent is confidently wrong. Self-review draws its signal from the same model: in the preliminary study of Section 3.2, it catches 50.3% of failing states, against 96.9% for an independent check.

Independent context. Chain-of-Verification (Dhuliawala et al., 2024) separates verification from the draft it verifies by answering its verification questions without the draft in context. SIDECAR keeps the separation, never reads the agent's conversation, and takes its answers from checks executed in the environment rather than from the model's own knowledge, as described in Sections 4.1 and 4.2.

A.3 TEST-TIME VERIFICATION

Test-time verification methods differ in where the corrective signal comes from and when it is applied, as summarized in Table 2. Methods that act after the run select among or revise finished outputs; LLM-as-a-Verifier (Kwok et al., 2026), for example, lifts TERMINAL-BENCH pass@1 by three points by selecting among finished runs. Among methods that act during the run, confidence-triggered sampling and imagined rollouts take their signal from the agent's model or from a world model that shares its priors, and tree search takes it from the environment but needs a resettable environment and multiplies the compute per task. SIDECAR shares its timing with runtime guards and monitors.

Table 2: Test-time reliability methods by whether their corrective signal comes from a model's judgment or from tools and the environment, and by when they act: after the run, on a finished output or attempt, or during the run, between the agent's steps.

	After the run	**During the run**
Model judgment	self-consistency, best-of-N, PRMs, generative and LLM verifiers (Wang et al., 2023; Snell et al., 2024; Lightman et al., 2024; Zhang et al., 2025a; Kwok et al., 2026); Self-Refine (Madaan et al., 2023); Reflexion, which carries reflections into the next attempt (Shinn et al., 2023)	confidence-triggered sampling, parallel self-reasoning (Lee et al., 2026b; Sun et al., 2026); imagined rollouts in a model-based world model (Gu et al., 2025); runtime guards and monitors (Fang et al., 2025; Lee et al., 2026a; Zheng et al., 2026; Naik et al., 2026; Liu et al., 2026); trusted monitoring (Greenblatt et al., 2024; Bhatt et al., 2025)
Tools or environment	tool-checked revision (Gou et al., 2024); generated tests, discussed in Appendix A.5	tree search in a resettable environment (Koh et al., 2024; Zhou et al., 2024a; Yu et al., 2025); **SIDECAR** (executable checks in its own context, raw check output)

Runtime guards and monitors. GuardianAgentBench (Naik et al., 2026) intercepts tool calls on business tasks and rescues 30 of 151 failing scenarios while blocking 2 of 429 passing ones; LivePlan (Liu et al., 2026) monitors a SWE-agent for plan drift, and in its per-instance transition matrix every intervention, including its own, broke some previously passing runs; StepGuard (Zheng et al., 2026) reports aggregate utility whose sign differs across its three benchmarks. Trusted monitoring in AI control (Greenblatt et al., 2024; Bhatt et al., 2025) likewise measures the safety a monitor adds together with the usefulness its interventions cost. Because an intervention can also break passing work, our end-to-end comparisons in Section 6.1 report paired losses together with paired wins.

Model-written feedback is the weak link. Olausson et al. (2024) find self-repair gains modest and bottlenecked by the quality of the model's feedback, with human feedback on the same failures far more effective, and CriticGPT (McAleese et al., 2024), a separately trained code critic, reports hallucinated problems as its main failure. On FeedbackEval (Dai et al., 2026), LLM-written feedback produced without access to the reference solution scores below a content-free "the code is wrong" message for all five repair policies. In AI control, Lucassen & Kaufman (2026) isolate the handoff channel: returning the monitor's written rationale to the untrusted model erases the safety gain that a rationale-free retry retains, because the model uses the rationale to route around the monitor. Their setting is adversarial; our benign captured-state comparison points the same way: the agent solves 21 of 32 states after the raw diagnostic and 10 after a model-written explanation, as detailed in Appendix B.4.2. By design, SIDECAR returns the kind of feedback used in conversational repair (Xia & Zhang, 2024), a failing assertion with its observed output, from a check that the verifier wrote and the framework executed, as described in Section 4.2.

Measuring an intervention. Evaluation noise on model benchmarks is large, and paired comparisons on the same questions reduce it (Miller, 2024); the LivePlan authors call for repeated paired rollouts. Lucassen & Kaufman (2026) and Iscan (2026) use paired statistics; the latter, a preregistered

study on small code models, compares raw execution counterexamples with resampling without feedback at a matched output budget. The captured-state experiments of Section 3.2 likewise pair two continuations of each rejected state, and the end-to-end comparisons of Section 6.1 pair runs of the same task and seed and apply an exact sign test to the discordant pairs.

A.4 Self-Improving Agents

Self-improving agents differ in where the improvement signal comes from, what it changes, and how far the change carries. The signal is usually experience from earlier trials, stored as reflections, skills, workflows or a knowledge base (Shinn et al., 2023; Wang et al., 2024; 2025; Tang et al., 2026); skills can also be distilled from code repositories (Chen et al., 2026a), and StateM starts from postmortem findings (Qin et al., 2026). What changes is the agent's context, its skill library or its harness (Gu, 2026; Li et al., 2026a). Most of these changes carry across trials or tasks, so what enters a persistent store decides what later runs inherit; this is where the risks of poisoned skills and unintended drift discussed in Section 2 arise.

SIDECAR takes a different position on each axis. Its signal is the execution feedback of its own checks in the current task. It changes only the executable logic of its checks, while claims, counterexamples and the framework's views, isolation, limits and budgets stay fixed, and a revised check is accepted only if it still returns FAIL on every counterexample of its claim, as described in Section 4.3. The change stays within one task run: notes start empty for each run, as specified in Appendix C.4. A revision therefore changes how a requirement is tested but not what the verifier may access or what its results authorize.

A.5 Generated Tests and Rubrics

The generated tests and rubrics discussed in Section 2 check a produced result, such as a program, a patch, or an answer. Verification without an oracle is the oracle problem (Barr et al., 2015): a check tests necessary conditions or consistency relations of the intended behavior, often through a generated test. Generated tests select code (Chen et al., 2023), drive iterative repair (Ridnik et al., 2024) or select among candidate patches (Xia et al., 2024), and can be trained to expose errors (Prasad et al., 2025); writing a test from an issue is itself a benchmarked skill (Mündler et al., 2024), as is turning intent into a postcondition (Endres et al., 2024). Executable verifiers can be synthesized and refined against labeled model outputs (Pezeshkpour & Hruschka, 2026), a research agent's answers can be checked against rubrics derived from a failure taxonomy (Wan et al., 2026), and candidate answers from sampled search trajectories can be scored against sub-questions decomposed from the question (Zhao et al., 2026a). For coding agents, Wang et al. (2026a) argue that verification is becoming the harder problem and that every buildable verifier is a proxy for intent.

SIDECAR runs such checks inside a long-horizon run: each check tests a claim that quotes a public requirement, and the check is rerun on the current state at every decision point whose committed result its claim concerns, so a check written early keeps testing the later work built on that result, as described in Section 4.2; in our experiments, described in Section 5, the decision points are the agent's confirmed completions.

B Preliminary Experiments

This appendix gives the protocols behind the four preliminary experiments in Section 3.2 and a supplementary comparison of raw output with a model-written explanation. The captured-state experiments use task-specific checks developed before evaluation and then frozen. They study detection, feedback and check revision separately from the end-to-end SIDECAR evaluation in Section 6.

B.1 Tasks, Captured States and the Predicate Φ

Tasks and captured states. The captured-state experiments use three TERMINAL-BENCH 2.0 tasks, `custom-memory`, `headless` and `polyglot`, with TERMINUS-2 agents using gpt-oss-120b, gpt-oss-20b, Qwen3-32B or Qwen3-Coder-30B. A captured state is the environment saved at the decision point where the agent confirms completion. In a paired continuation, a rejected state is

Table 3: Public requirements and predicate stages in the captured-state experiments. A state passes Φ only if every stage passes.

Task	Public requirements	Stages of Φ
`custom-memory`	Build in release and debug, run natively under both, and run clean under Valgrind against an unmodifiable `main.cpp` with a custom heap	Build release; build debug; run both natively; Valgrind clean
`headless`	Import and instantiate `HeadlessTerminal`, satisfy its public base-class interface, accept keystrokes and handle Ctrl-C	Import; interface; instantiation; keystrokes; Ctrl-C
`polyglot`	Run the same file under `python3` and compile it under `gcc`, with both halves printing $f(N)$	Source present; Python run; C compile; C run and compare

resumed twice from the same environment; a later confirmed completion is accepted without another check. This isolates a single feedback intervention rather than the repeated verification of SIDECAR.

Outcome predicate. Each task has a predicate Φ authored from its public requirements, independently of the agent's own assessment and the hidden benchmark tests. It evaluates the necessary conditions in Table 3 in order and reports the first failing stage. A state is called solved in these experiments when it passes Φ; passing these necessary conditions does not establish that it passes all official tests. A hanging native program is labeled failing on `custom-memory`, while a hanging instantiation on `headless` has no verdict and is excluded.

Scope and development exposure. The task-specific checks were selected using development states labeled by Φ, as described in Appendix B.5. At least two development states also occur among the states evaluated in Experiments 2 and 3, so these observations are not wholly independent of development.

Execution validity. A check execution is valid when it exits with code 0 and its standard output contains exactly one verdict token. Output after the token does not invalidate the verdict, although the generator prompt in Appendix B.6 asks for the token on the final non-empty line; an execution without a valid verdict does not reject the state.

B.2 Experiment 1: The Agent's Own Review

This experiment supports the finding in Section 3.2 that the agent's own review accepts most of its failing work. It runs gpt-oss-120b with TERMINUS-2 and no verifier over three seeds. Before accepting completion, TERMINUS-2 asks the agent to confirm its earlier completion proposal, giving it another opportunity to inspect its work in its own context. Across these runs, 87% of the failing runs end with the agent confirming completion. In 10 of 21 annotated failing runs, the output revealing the failure was already in the agent's context when it confirmed. This native confirmation step differs from the explicit self-review of Experiment 2, which is measured on captured states.

B.3 Experiment 2: Failure Detection

This experiment supports the finding in Section 3.2 that executable checks in a separate context detect nearly all failing states. For each task, gpt-oss-120b writes a check in a context separate from the agent's. The check is developed on labeled states, frozen, and then executed in a network-disabled disposable copy of each captured environment, without an additional model call. Only a FAIL verdict rejects completion. For self-review, the agent that produced each failing gpt-oss-120b state is explicitly asked, in its own context, to check its work, and it catches the failure when it rejects completion.

Figure 1a reports the comparison on gpt-oss-120b states: self-review catches 50.3% of the failures and the independent check 96.9%.

B.4 Experiment 3: Recovery after Rejection

This experiment supports the finding in Section 3.2 that returning evidence at a decision point helps the agent recover. Each rejected state is continued twice from the same environment, once after a plain rejection and once after a rejection that carries a diagnostic, and Φ judges each final state.

B.4.1 Plain Rejection versus Diagnostic Feedback

Diagnostic selection. Fixed code selects one diagnostic for each rejected state. On `headless` and `polyglot`, it runs the public-requirement stages and returns the output of the first failure. On `headless` it also reports the class's method-resolution order, whether it is abstract, its abstract methods and the relevant method signature. On `custom-memory`, a build failure returns the compiler output already produced by the frozen check; a native failure triggers an AddressSanitizer rebuild and returns its trace. The returned evidence comes from these task-specific diagnostics. Output is retained up to 6000 characters, with truncation explicitly marked.

Feedback messages and continuation. The plain rejection says that the task is not complete and asks the agent to recheck it and continue if necessary, without a diagnostic. The diagnostic condition adds a fixed preamble saying that a separate verifier rejected completion, followed by the diagnostic name, command, return code, stdout, stderr and available introspection. Both continuations use the same backbone and per-turn seed sequence, with a maximum of eight turns.

Paired outcomes. The 128 pairs in Figure 1b cover three tasks and two backbones, gpt-oss-120b and Qwen3-Coder-30B. The task and backbone combinations in this pool were selected after their outcomes were seen, so the reported rates describe this selected pool. After the plain rejection, the agent solves 34 states (26.6%); with the diagnostic, it solves 63 (49.2%). The diagnostic gains 35 states that the plain continuation fails to solve and loses 6 that the plain continuation solves.

Intervention on passing work. This test measures the harm of an unnecessary rejection to correct work. Twelve `custom-memory` / gpt-oss-120b states that both Φ and the check passed were rejected once with the plain message and continued under the same eight-turn cap. All twelve still passed Φ afterward.

A harm mechanism on `polyglot`. A Python traceback can point to a C line that the polyglot program needs, even when the defect is in the upstream delimiter that should hide that line from Python. In two inspected states, the C half compiled before the rejection but failed to compile after the agent acted on the Python diagnostic. Thus raw evidence can also direct an unsuccessful repair. This diagnosis is limited to those two states of one task.

B.4.2 Raw Output versus a Model-Written Explanation

This supplementary comparison examines the feedback choice discussed in Section 4.2. On 32 rejected `headless` / gpt-oss-120b states, a third continuation, alongside the plain and diagnostic ones, receives an explanation of the diagnostic written by a separate gpt-oss-120b call, the note writer. It uses the same turn cap and per-turn seeds as the other continuations. The agent solves 15 states with plain rejection, 21 with the diagnostic, and 10 with the explanation. The diagnostic condition wins 14 paired outcomes against the explanation condition and loses 3.

The explanation condition adds the note writer's four fields but omits the original command, return-code and raw-introspection blocks. It therefore changes both the interpretation and the available raw information. The experiment compares the resulting feedback packages.

An example feedback packet. For a `headless` state that fails the interface stage, the two conditions carry the following blocks after their fixed preambles. The explanation condition renders the note writer's fields above the name of the diagnostic, the raw stdout and the raw stderr.

Diagnostic condition: raw output

```
Fixed primitive: interface_traceback

Command:
python3 -c <public interface check>

Return code: 1

Raw stdout:


Raw stderr:
Traceback (most recent call last):
  File "<string>", line 6, in <module>
    assert not inspect.isabstract(HeadlessTerminal)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError

Raw introspection:
mro: ['HeadlessTerminal', 'BaseTerminal', 'ABC', 'object']
abstract: True
abstract_methods: ['send_keystrokes']
send_keystrokes: (self, keystrokes: str, wait_sec: float = 0.0) -> None
```

Explanation condition: note writer's fields

```
Diagnosis status: localized
Established error: HeadlessTerminal remains abstract because it does
not implement the required abstract method `send_keystrokes`.
Location: /app/headless_terminal.py
Evidence:
- AssertionError
- abstract: True
- abstract_methods: ['send_keystrokes']
- return_code: 1
```

B.5 Experiment 4: Check Revision from Execution Feedback

This experiment gives the controlled measurement summarized in Section 3.2: a check is revised from its execution feedback on development states labeled by Φ and then evaluated on held-out states.

Development procedure. The check generator runs a generate, execute and revise loop on gpt-oss-120b development states labeled by Φ. Each revision receives the previous check and its execution outputs together with the states' labels. The first candidate that rejects the failing development states without rejecting passing ones is frozen for evaluation.

Controlled comparison. Figure 1c compares the initial `headless` check with the frozen revised check on 59 held-out states. Both reject the 19 failing states. The initial check also rejects all 40 passing states, while the revised check rejects none of them, raising correct judgments from $19/59$ to $59/59$. All executions return a verdict, so the change concerns false rejections rather than the ability to execute.

The initial check imports `BaseTerminal` as a separate class object instead of taking the base class that `HeadlessTerminal` inherits, so its interface test fails even on a correct implementation and it rejects all 59 states. The revised check obtains the base class from the method-resolution order. Revision also drops tests of output and process liveness. For example, a constructed state in which `send_keystrokes` is correctly inherited but does nothing passes the revised check, although a behavioral test that sends keystrokes rejects it.

B.6 Check Generator and Note Writer

These prompts are used in the captured-state experiments. The check generator is gpt-oss-120b for every task, including checks applied to other agent backbones. The note writer also uses gpt-oss-120b.

Check generator. The generator writes a reusable check from a public requirement. Revision inputs include the previous source, each development state's Φ label, return code, stdout, stderr and verdict. The model does not receive the implementation of Φ, hidden benchmark

tests or reference solutions. The source is frozen after development. The system prompt follows, with its verdict tokens written as PASS, FAIL and UNKNOWN.

Check generator: system prompt

```
You are improving a verifier after receiving delayed objective feedback. Write one reusable,
read-only Bash detector for the stated public necessary condition. It will run from /app in a
network-disabled disposable clone of a future unseen state. It must inspect the actual state,
exit zero, and print exactly one of PASS, FAIL, or UNKNOWN as its final non-empty line. It may
print evidence first. Do not call benchmark tests, reward, an official verifier, a reference
solution, or network services. Do not hardcode state IDs or labels. Return only the requested
JSON.
```

Note writer. The note writer reads the diagnostic and returns a structured explanation. It uses reasoning effort `medium`, temperature 0 and a 16,384-token output limit, reading at most 16,000 characters of the diagnostic. Its system prompt is:

Note writer: system prompt

```
You are a clean-context verifier. Code has already chosen and run a fixed diagnostic primitive
based only on the externally observed failure stage. Summarize what the raw evidence establishes
for the policy agent. Do not suggest a repair. Do not turn a symptom into an unsupported unique
cause. If the causal fault is not localized, say unresolved and state the narrowest established
failure location. Return one JSON object with exactly these keys: diagnosis_status,
error_claim, location, evidence. diagnosis_status is localized or unresolved; evidence is a
nonempty list of short verbatim facts from the supplied output.
```

C Experimental Details

This appendix gives the details behind Sections 5 and 6: the benchmark scope, the agent settings, the decision points of the Baseline and the two controls, the configuration of SIDECAR, the metrics and statistics, the complete success rates and the definitions of trial outcomes.

C.1 Benchmarks and Task Scope

TERMINAL-BENCH. The TERMINAL-BENCH 2.0 results use the 84 tasks retained from the 89-task benchmark after excluding `break-filter-js-from-html`, `vulnerable-secret`, `feal-differential-cryptanalysis`, `feal-linear-cryptanalysis` and `crack-7z-hash`. GPT-6-Astra declines these five offensive-security tasks under its safety policy, so they are excluded for every backbone. Tasks run in separate containers and are scored by their official tests after the agent finishes. The TERMINAL-BENCH 2.1 comparison in Figure 3 uses the same 84 task names, with the task revisions supplied by version 2.1; the leaderboard entries in that figure are recomputed on this task set.

SWE-bench Verified. We use the 500 instances of `swebench-verified@1.0` in HARBOR. The agent receives the issue text and edits the repository in `/testbed`. After the agent finishes, the official grader restores the test files touched by the hidden test patch to the base commit, applies that patch, runs the instance's test command and parses its output with `swebench` 4.0.3. A pass requires all fail-to-pass and pass-to-pass tests to pass.

Development scope. Development included the captured-state tasks in Appendix B and separate benchmark smoke tests, including `regex-log` on TERMINAL-BENCH and `django__django-11179` on SWE-bench Verified. These tasks remain in the reporting scope; the benchmark results are not restricted to tasks unused during development.

C.2 Agent Settings, Resources and Serving

The experiments use HARBOR 0.18.0 and its TERMINUS-2 scaffold. Table 4 gives the agent settings. Each task keeps its configured agent time limit, including time spent on verification; the time limit is not extended for an intervening system. SWE-bench Verified uses a 3,000 s agent limit.

The open-weight backbones are served locally with vLLM using tensor parallelism 2 on NVIDIA RTX PRO 6000 Blackwell Server Edition GPUs. GPT-6-Astra is accessed through a hosted OpenAI-compatible endpoint. Within a backbone comparison, every model call, including

Table 4: Agent settings for the comparisons in Section 6.

Setting	Value
Scaffold	TERMINUS-2 in HARBOR 0.18.0
Maximum turns	200
Temperature, top-p, reasoning effort	0.7, 1.0, medium
Response parser, API	JSON, chat completions
Task container	Docker, 4 CPUs, 8,192 MB memory
Agent time limit	Per task; 3,000 s on SWE-bench Verified

those of the verifier and of Self-reflection, uses that backbone. Task containers are independent even when trials run concurrently.

C.3 DECISION POINTS, BASELINE AND CONTROLS

Confirmed completions. In native TERMINUS-2, a first `task_complete` proposal marks the task as pending completion and asks the agent to confirm. A second consecutive proposal ends the task after executing any commands in that turn. We call this second proposal a *confirmed completion*; confirmed completions are the decision points of all four systems. The Baseline uses this native loop without interception or added feedback.

Table 5 summarizes how the four systems act at a confirmed completion; the paragraphs below give the exact rules.

Table 5: The four systems at a confirmed completion.

System	What it examines	When it rejects	What the agent receives
Baseline	Nothing	Never	Nothing; the task ends
Rejection rule	Remaining time and a file-state fingerprint; no model call	While at least one third of the time remains: at the first confirmation, and at later ones if files changed since the previous confirmation	A generic message
Self-reflection	The agent's full conversation, reread by the agent's backbone	When the backbone judges a requirement unmet or not yet verified	The backbone's reflection
SIDECAR	Executable checks, written by the verifier in its own context and run on a copy of the current environment	When an accepted check returns FAIL	The unedited output of the failing check

Handling the confirming turn. Self-reflection and the Rejection rule inspect confirmed completions and discard any commands in a turn they reject. SIDECAR first executes the commands carried by a confirmation, returns their output and resets pending completion. The agent then confirms again after seeing the result; a confirmation without commands is the decision point at which SIDECAR runs its checks. Executing the pending commands is not counted as a rejection. All three intervening systems keep the task incomplete when they reject, and permit at most four rejections per task.

Self-reflection. At a confirmed completion, Self-reflection calls the agent's backbone with a copy of its current conversation and the following review request:

Self-reflection: review request to the agent's backbone

```
Before this task is accepted as complete, review your own work.
Re-read the task description and list every explicit requirement.
For each requirement, decide from what you have already observed in this conversation whether it
is verified to be met.
Reply with only a JSON object:
{"complete": true or false, "reflection": "which requirements are not yet verified or not met,
and what to do next"}
```

A reply with `complete=false` rejects completion and returns `Completion rejected by self-review:` followed by the reflection, limited to 2,000 characters. A reply with

`complete=true` accepts it. A reply that does not parse as JSON is retried once; a second parse failure accepts completion. After four rejections, completion is accepted without another review.

Rejection rule. The Rejection rule makes no model call and performs no correctness check. It rejects the first confirmed completion if at least one third of the agent's time budget remains. At later confirmations it accepts when less than one third remains, when the file-state fingerprint has not changed since the preceding confirmation, or after four rejections; otherwise it rejects with `completion rejected; continue`. The fingerprint uses paths, sizes and modification times of changed ordinary files, excluding temporary files, caches and logs. It does not hash file contents. If the timer or fingerprint is unavailable, the rule accepts completion.

C.4 Configuration of Sidecar

Following Section 4, the verifier writes checks, while the framework controls their observations, execution, acceptance and effect on the agent.

Table 6: Principal limits in the SIDECAR specification. The framework fixes these limits within a task.

Item	Limit
Claims	At most 4, written before the agent's first action
Requirement quotation	Literal task text, at most 1,024 characters after whitespace normalization
Check source	4,096 bytes before framework wrapping
Check execution	At most 60 s, subject to remaining verification time
Check sandbox	4 CPUs, 8,192 MB memory, no network; discarded after execution
Rejection evidence	Failed-assertion observations, stdout and stderr, each bounded to 2,048 bytes
Rejections	At most 4 per task
Verification time	Cumulative target of 10% of the agent's time limit
Verifier sampling	Temperature 0, reasoning effort medium

Inputs, checks and notes. The verifier receives the public task, observations of the environment and its own notes, without the agent's conversation. Before the agent acts, it derives claims from the task and writes each claim's first check and counterexamples; each check then runs on the initial environment. Checks use the three views of Section 4.2: file listings, file contents and command output. They execute in disposable sandboxes of the current environment state, preserving the agent's workspace. For each claim, the notes store its quoted requirement, counterexamples, check versions, execution results and outputs. Notes start empty for each task run.

Results and acceptance. The framework records a check's assertions through `expect` and its prerequisites through `require_ready`. An unmet assertion produces FAIL; an unmet prerequisite produces UNKNOWN. A missing checker dependency, an execution error or a timeout also produces UNKNOWN. A missing deliverable named by the claim is a failed requirement, rather than a checker prerequisite. The framework validates the source and its requirement quotations before execution. For acceptance, the check must return a usable verdict on the current state and FAIL on every counterexample, including the standard deletion and emptying edits for named files. An edit that cannot be applied does not count as a successful counterexample test. A separate model review at acceptance uses these outputs to assess whether the check measures its claim.

Revision and reuse. A source-validation error, a failed acceptance test or a fault exposed by execution can prompt revision. The verifier receives the fixed claim, current source and relevant feedback, and changes only the check code. Each new or revised check must pass acceptance before it is retained. Retained checks are rerun on the current state at later decision points without a model call for the rerun itself. Revision and acceptance review are separate model calls and use the same verification-time budget. Checks that have not passed acceptance do not reject completion.

Feedback and budgets. When a retained check fails and rejection budget remains, the framework keeps the task incomplete and returns its execution evidence without an additional model explanation. The evidence contains failed-assertion observations and captured stdout and stderr, mechanically bounded as in Table 6. An UNKNOWN result alone cannot reject completion. Completion is accepted after four rejections or once less than 1 s of verification budget remains.

The cumulative budget covers verification work within the agent's unchanged time limit; operations already in flight can finish after the target is reached.

C.5 Scoring, Metrics and Statistics

Task success and paired comparisons. A passing trial has official reward 1. Unless otherwise specified in a figure caption, success rate is the fraction of passing trials among those with an official reward in the reported comparison. The paired comparisons count wins and losses on matched trials of the same task, backbone and seed: a win means only SIDECAR passes, and a loss means only the Baseline passes. The exact two-sided sign test uses these discordant pairs and ignores ties. The pooled tests use trial pairs as observations and do not adjust for dependence between repeated measurements of the same task. Appendix C.6 collects the counts reported in Section 6.1.

Check lifecycle. Figure 7 follows the gpt-oss-120b checks on TERMINAL-BENCH 2.0. Its first two bars count checks run on the initial environment, before the agent acts: *initial* records each check's first execution, and *after revision* records the execution of its latest version, retaining the first result for a check that was not revised. The final bar instead counts executions of retained checks at decision points. A *usable verdict* is PASS or FAIL. These measures quantify verdict availability rather than correctness or task success.

Cost. Section 6.4 reports model calls, tokens and verification time for gpt-oss-120b on TERMINAL-BENCH 2.0 and SWE-bench Verified. Agent calls and verifier calls are counted separately; execution of a stored check is not a model call. Verification-time share is the cumulative verification time divided by the trial's wall-clock time, summarized by its median across trials. The budget target instead uses the task's maximum allowed agent time. The timeout frequency is the share of trials that end at the agent time limit.

C.6 Complete Results

Table 7 collects the success rates in Figures 3, 4 and 5. The comparison with the two additional controls is shown in Figure 6: on TERMINAL-BENCH 2.0 with gpt-oss-120b, the Baseline scores 17.3%, Self-reflection 21.7%, the Rejection rule 23.4% and SIDECAR 25.0%. Their mean rejections per trial are 1.38 for Self-reflection, 1.66 for the rule and 0.30 for SIDECAR.

Table 7: Task success rates (%) reported in the main figures. The TERMINAL-BENCH comparisons use the 84-task reporting scope. The TERMINAL-BENCH 2.1 row counts unscored trials as failures, as in Figure 3; the other rows are computed over the trials with an official reward.

Benchmark	Backbone	Baseline	SIDECAR
TERMINAL-BENCH 2.0	gpt-oss-20b	5.6	9.7
	gpt-oss-120b	17.3	25.0
	GPT-6-Astra	77.1	80.7
TERMINAL-BENCH 2.1	GPT-6-Astra	79.8	84.5
SWE-bench Verified	gpt-oss-20b	4.2	6.2
	gpt-oss-120b	11.6	15.1
	GPT-6-Astra	83.8	83.8

Across the three backbones on TERMINAL-BENCH 2.0, the comparison with the Baseline has 40 discordant pairs: 30 wins and 10 losses for SIDECAR ($p = 0.002$). On SWE-bench Verified, the corresponding 129 discordant pairs comprise 78 wins and 51 losses ($p = 0.022$).

C.7 Trial Outcomes

The agent's stopping decision and the official task outcome answer different questions. A *pass* has official reward 1. An *accepted failure* occurs when completion is accepted but the official tests fail. A *timeout failure* or *turn-limit failure* ends at the corresponding agent limit and fails the official tests. Acceptance therefore describes how the agent stopped, rather than certifying that the task was solved.

Section 6.2 concerns accepted failures: most failing runs end with the agent confirming completion on work that fails, and rejecting that completion lets the agent continue, so the continuation can

end in a pass, in another accepted failure, or at the agent's time or turn limit. Section 6.4 concerns trials that end at the time limit: verification time counts against the agent's unchanged time limit, so the agent reaches the limit more often, even in trials where no completion is rejected.

D CASE STUDIES

Each case below reproduces the records of one trial in the order in which they were produced: the task text, what the verifier wrote, what the framework returned, what the agent did, and the official result. Box titles name the source of each record: the **task**, the **verifier**, the **framework**, the **agent** and the **official tests**. The contents are copied from the run logs; check results are shown without run identifiers and hashes, and [...] marks lines removed for length.

D.1 VERIFICATION AND RECOVERY

Task `build-pov-ray` on TERMINAL-BENCH 2.1, run by GPT-6-Astra with SIDECAR. Before the agent's first action, the verifier reads the task and writes one claim, with the quoted requirement, the file it concerns and one counterexample; the framework adds its two standard counterexamples for that file.

Task: public task description

```
Build POV-Ray 2.2. Find and download the source archives, extract them to `/app/povray-2.2`, then
    ↪  compile and install to `/usr/local/bin/povray`.

We will test your build by rendering `/app/deps/illum1.pov` and comparing against a reference
    ↪ image.
Do not modify `/app/deps/illum1.pov`.

As a sanity check to see if your build works, you can run `/usr/local/bin/povray +L/app/povray
    ↪ -2.2/povdoc/include +I/app/deps/illum1.pov +O/dev/null +P -V`. This should complete
    ↪ successfully and show rendering statistics.
```

Verifier: claim and counterexamples

```
{
  "claim": {
    "files": [
      "/usr/local/bin/povray"
    ],
    "quote": "As a sanity check to see if your build works, you can run `/usr/local/bin/povray +L
     ↪ /app/povray-2.2/povdoc/include +I/app/deps/illum1.pov +O/dev/null +P -V`. This should
     ↪ complete successfully and show rendering statistics.",
    "statement": "The installed executable completes the specified sanity-check render with exit
     ↪ status zero and emits numerical rendering statistics."
  },
  "counterexamples": [
    {"new": "", "old": "", "operation": "empty", "origin": "model", "path": "/usr/local/bin/
     ↪ povray"},
    {"new": "", "old": "", "operation": "delete", "origin": "standard", "path": "/usr/local/bin/
     ↪ povray"},
    {"new": "", "old": "", "operation": "empty", "origin": "standard", "path": "/usr/local/bin/
     ↪ povray"}
  ]
}
```

Verifier: accepted check

```
import re

def check():
    prerequisites = observe_command(['/bin/sh', '-c', 'if test -r /app/deps/illum1.pov; then
     ↪ printf "scene-readable\\n"; fi; if command -v tail >/dev/null 2>&1; then printf "tail-
     ↪ available\\n"; fi'], timeout=10)
    require_ready('scene-readable' in prerequisites and 'tail-available' in prerequisites,
     ↪ prerequisites)
    output = observe_command(['/bin/sh', '-c', '{ /usr/local/bin/povray +L/app/povray-2.2/povdoc/
     ↪ include +I/app/deps/illum1.pov +O/dev/null +P -V </dev/null; status=$?; printf "\\
     ↪ nCHECK_RENDER_EXIT=%s\\n" "$status"; } 2>&1 | tail -c 7500'], timeout=60)
    expect('CHECK_RENDER_EXIT=0\n' in output, output, requirement='As a sanity check to see if
     ↪ your build works, you can run `/usr/local/bin/povray +L/app/povray-2.2/povdoc/include +I/
     ↪ app/deps/illum1.pov +O/dev/null +P -V`. This should complete successfully and show
     ↪ rendering statistics.')
    statistics = re.search(r'(?im)^.*(?:pixels|rays|samples|intersection tests).*\d.*$', output)
    expect(statistics is not None, output, requirement='As a sanity check to see if your build
     ↪ works, you can run `/usr/local/bin/povray +L/app/povray-2.2/povdoc/include +I/app/deps/
     ↪ illum1.pov +O/dev/null +P -V`. This should complete successfully and show rendering
     ↪ statistics.')
```

The agent downloads the sources, compiles them, installs the program and runs the sanity command itself. The command succeeds, and the agent confirms completion twice, as TERMINUS-2 requires.

Agent: own sanity render and completion

```
Analysis: POV-Ray 2.2 compiled successfully with only a statistics-format warning.
Plan: Install the executable and run the requested sanity render without modifying the scene.
[keystrokes] install -m 755 povray /usr/local/bin/povray; cd /app; /usr/local/bin/povray +L/app/
    ↪ povray-2.2/povdoc/include +I/app/deps/illum1.pov +O/dev/null +P -V > /tmp/pov-sanity.log
    ↪ 2>&1; echo "Render exit status: $?"; tail -55 /tmp/pov-sanity.log
[terminal]
Render exit status: 0
[...]
/app/deps/illum1.pov statistics
---------------------------------------
Resolution 100 x 100
# Rays:       91599    # Pixels:       10000  # Pixels supersampled:          0
[...]

Analysis: POV-Ray 2.2 has been downloaded, extracted into /app/povray-2.2, compiled, and
    ↪ installed at /usr/local/bin/povray. The exact requested sanity render completed with exit
    ↪  status 0 and reported statistics for 10,000 pixels and 91,599 rays. /app/deps/illum1.pov
    ↪  was not modified.
Plan: No further steps are required. The source archives remain in /app/archives, and the sanity-
    ↪ render log is /tmp/pov-sanity.log.
[tool call] mark_task_complete

Analysis: The build and installation are complete. The requested sanity check exited successfully
    ↪  with rendering statistics, and the input scene was left unchanged.
Plan: Submit the completed build for grading.
[tool call] mark_task_complete (confirmation)
```

At this confirmed completion the framework reruns the check on a disposable copy of the current state. The check fails, and the framework returns its output to the agent in place of ending the trial. The box shows the agent's next observation as it was received:

Framework → agent: rejection with the check's output

```
Previous response had warnings:
WARNINGS:
  Persistence of Vision Raytracer Ver 2.2.u
    This is an unofficial version compiled by:
     FILL IN NAME HERE.........................
     The POV-Ray Team is not responsible for supporting this version.
  Copyright 1993 POV-Team
  ------------------
scene-readable
tail-available

  Persistence of Vision Raytracer Ver 2.2.u
    This is an unofficial version compiled by:
     FILL IN NAME HERE.........................
     The POV-Ray Team is not responsible for supporting this version.
  Copyright 1993 POV-Team
  --------------------------------------------------------------------------
  POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A. Collins.
    Contributing Authors: (Alphabetically)
       Steve Anger        Steve A. Bennett   David K. Buck
       Aaron A. Collins   Alexander Enzmann  Dan Farmer
       Douglas Muir       Bill Pulver        Robert Skinner
       Scott Taylor       Drew Wells         Chris Young
    Other contributors listed in the documentation.
  --------------------------------------------------------------------------
Parsing...Cannot open include file: shapes.inc

POV-Ray Options in effect: +fd +p +mb25 -q9 -w100 -h100 -s1 -e100
-k0.000 -mv2.0 -i/app/deps/illum1.pov -o/dev/null -l/app/povray-2.2/povdoc/include

CHECK_RENDER_EXIT=1



New Terminal Output:

root@15c169648628:/app#
```

Agent: diagnosis

```
Analysis: The latest check failed to locate shapes.inc despite the earlier successful render, so
     ↪ completion needs to be rechecked.
Plan: Verify the include files and repeat the exact sanity command to determine whether the
     ↪ failure is reproducible.
[keystrokes] ls -ld /app/povray-2.2/povdoc/include; ls -l /app/povray-2.2/povdoc/include/shapes.
     ↪ inc; /usr/local/bin/povray +L/app/povray-2.2/povdoc/include +I/app/deps/illum1.pov +O/dev
     ↪ /null +P -V >/tmp/pov-recheck.log 2>&1; echo EXIT=$?; tail -30 /tmp/pov-recheck.log; grep
     ↪  -n 'Library\|Library_Path\|strcat.*file\|FILENAME_SEPARATOR' /app/povray-2.2/build/{
     ↪ tokenize.c,frame.h,povray.c} | tail -35
[terminal]
drwx------ 2 238 103 4096 Oct 11  1993 /app/povray-2.2/povdoc/include
-rw------- 1 238 103 3238 Oct 11  1993 /app/povray-2.2/povdoc/include/shapes.inc
EXIT=0
[...]
```

Agent: repair and new completion

```
Analysis: The archives preserved restrictive permissions (directories 700 and files 600 owned by
     ↪ UID 238). Root can render, but an unprivileged grader cannot read the include files.
Plan: Make extracted files readable and directories traversable, then validate the render as
     ↪ nobody.
[keystrokes] chmod -R a+rX /app/povray-2.2; runuser -u nobody -- /usr/local/bin/povray +L/app/
     ↪ povray-2.2/povdoc/include +I/app/deps/illum1.pov +O/dev/null +P -V > /tmp/pov-
     ↪ unprivileged.log 2>&1; echo CHECK_RENDER_EXIT=$?; tail -24 /tmp/pov-unprivileged.log
[terminal]
CHECK_RENDER_EXIT=0
[...]
/app/deps/illum1.pov statistics
---------------------------------------
Resolution 100 x 100
# Rays:       91599    # Pixels:       10000  # Pixels supersampled:          0
[...]

Analysis: The include-file permissions are fixed. The requested sanity render now succeeds as the
     ↪  unprivileged nobody user with exit status 0, reporting 10,000 pixels and 91,599 rays.
     ↪ POV-Ray 2.2 is installed at /usr/local/bin/povray, the sources and includes are in /app/
     ↪ povray-2.2, and illum1.pov was not modified.
Plan: The build is ready for grading; no further changes are needed.
[tool call] mark_task_complete
```

At the new completion the framework reruns the same check, which passes, and the trial ends.

Framework: check result at the second completion


```
{
  "status": "pass",
  "execution_error": "",
  "observations": [
    {
      "satisfied": true,
      "observation": "named file /usr/local/bin/povray exists and is nonempty: True",
      "requirement": "As a sanity check to see if your build works, you can run `/usr/local/bin/
     ↪ povray +L/app/povray-2.2/povdoc/include +I/app/deps/il"
    },
    {
      "satisfied": true,
      "observation": "\n  Persistence of Vision Raytracer Ver 2.2.u\n    This is an unofficial
     ↪ version compiled by:\n     FILL IN NAME HERE.....................\n     The POV-Ray
     ↪ Team is not responsible for supporting this version.\n  Copyright 1993 POV-Team\n
     ↪ -------------------",
      "requirement": "As a sanity check to see if your build works, you can run `/usr/local/bin/
     ↪ povray +L/app/povray-2.2/povdoc/include +I/app/deps/il"
    },
    {
      "satisfied": true,
      "observation": "\n  Persistence of Vision Raytracer Ver 2.2.u\n    This is an unofficial
     ↪ version compiled by:\n     FILL IN NAME HERE.....................\n     The POV-Ray
     ↪ Team is not responsible for supporting this version.\n  Copyright 1993 POV-Team\n
     ↪ -------------------",
      "requirement": "As a sanity check to see if your build works, you can run `/usr/local/bin/
     ↪ povray +L/app/povray-2.2/povdoc/include +I/app/deps/il"
    }
  ],
  "prerequisites": [
    {
      "satisfied": true,
      "observation": "scene-readable\ntail-available\n"
    }
```

```
  ]
}
```

Official tests

```
================================== PASSES ==================================
PASSED ../tests/test_outputs.py::test_illum1_render_and_verify
PASSED ../tests/test_outputs.py::test_povray_version
PASSED ../tests/test_outputs.py::test_povray_built_from_correct_source
============================== 3 passed in 9.38s ===============================
```

The paired Baseline trial on the same task and seed also ran the sanity command, observed exit status 0 and confirmed completion; it failed the official test `test_povray_built_from_correct_source`. This pair shows the two runs side by side; it does not show what the SIDECAR agent would have done without the rejection.

D.2 CHECK REVISION FROM EXECUTION FEEDBACK

Revision from an execution error. Task `regex-log` on TERMINAL-BENCH 2.0, run by gpt-oss-120b with an earlier configuration of SIDECAR that accepted a revised check without a counterexample test. This is the example of Section 3.2.

Task: public task description

````
Write a regex expression that matches dates in the format YYYY-MM-DD appearing in lines that
    ↪ contain an IPv4 address in a log file.
If multiple dates are present in a line, the regex should match only the last date in that line.
Assume that February can have up to 29 days in all years, without distinguishing leap years from
    ↪ non-leap years.
IPv4 addresses use normal decimal notation without leading zeros in each octet.

Note: Be careful that there might be text in the log that looks similar to dates or IPv4
    ↪ addresses but is not (e.g., user 1134-12-1234).
To avoid false matches, ensure that valid dates and IPv4 addresses are not immediately preceded
    ↪ or followed by alphanumeric characters.

Save your regex in /app/regex.txt
The regex will be read from the file and applied to the log file contents using Python's re.
    ↪ findall with the re.MULTILINE flag.
Example Python usage:
```
import re

with open("/app/regex.txt") as f:
    pattern = f.read().strip()

matches = re.findall(pattern, log_text, re.MULTILINE)
```
````

Verifier: first check

```
from pathlib import Path
import re

def check():
    # Load the submitted regex
    regex_path = Path('/app/regex.txt')
    expect(regex_path.is_file(), f'file {regex_path} exists={regex_path.is_file()}')
    pattern = regex_path.read_text().strip()
    # Compile to ensure it is a valid regex
    compiled = re.compile(pattern)

    # Helper to run a single test line and return the list of matches
    def run(line: str):
        return re.findall(compiled, line, re.MULTILINE)

    # 1. Line with IPv4 and two dates - only the last date should be returned
    line1 = '192.168.1.1 something 2023-05-12 other 2023-06-01'
    matches1 = run(line1)
    expect(matches1 == ['2023-06-01'], f'line1 matches={matches1}')

    # 2. Line without an IPv4 address - no matches
    line2 = 'no ip 2023-07-15'
    matches2 = run(line2)
    expect(matches2 == [], f'line2 matches={matches2}')
```

```
    # 3. Date immediately preceded by an alphanumeric character - should not match
    line3 = '10.0.0.1 abc2023-08-01'
    matches3 = run(line3)
    expect(matches3 == [], f'line3 matches={matches3}')

    # 4. Text that looks like a date but is not a valid month/day - should not match
    line4 = 'user 1134-12-1234'
    matches4 = run(line4)
    expect(matches4 == [], f'line4 matches={matches4}')

    # 5. IPv4 octet with a leading zero - should not be considered a valid IPv4, thus no match
    line5 = '01.2.3.4 2023-09-10'
    matches5 = run(line5)
    expect(matches5 == [], f'line5 matches={matches5}')

    # 6. Date with an invalid month (13) - should not match
    line6 = '10.0.0.5 2023-13-01'
    matches6 = run(line6)
    expect(matches6 == [], f'line6 matches={matches6}')
```

On the initial state, `/app/regex.txt` does not exist, so the check stops at its first assertion and never reaches the call to `re.findall`:

Framework: check result on the initial state


```
{
  "status": "fail",
  "execution_error": "",
  "observations": [
    {
      "satisfied": false,
      "observation": "file /app/regex.txt exists=False"
    }
  ]
}
```


At the agent's completion the file exists, the call is reached, and the check cannot reach a verdict:

Framework: check result at the agent's completion


```
{
  "status": "unknown",
  "execution_error": "ValueError: cannot process flags argument with a compiled pattern",
  "observations": [
    {
      "satisfied": true,
      "observation": "file /app/regex.txt exists=True"
    }
  ]
}
```


Framework → verifier: revision request, sent with the claim, the check and the task


```
{
  "operation": "repair execution error only; return exactly one check with identical claim",
  "execution_error": "ValueError: cannot process flags argument with a compiled pattern"
}
```


Verifier: revision, as a diff against the first check

```
--- check v0
+++ check v1
@@ -8,4 +8,4 @@
     pattern = regex_path.read_text().strip()
-    # Compile to ensure it is a valid regex
-    compiled = re.compile(pattern)
+    # Ensure the regex is syntactically valid
+    re.compile(pattern)

@@ -13,3 +13,3 @@
     def run(line: str):
-        return re.findall(compiled, line, re.MULTILINE)
+        return re.findall(pattern, line, re.MULTILINE)
```

Framework: revised check on the same state


```
{
  "status": "pass",
  "execution_error": "",
  "observations": [
    {
      "satisfied": true,
      "observation": "file /app/regex.txt exists=True"
    },
    {
      "satisfied": true,
      "observation": "line1 matches=['2023-06-01']"
    },
    {
      "satisfied": true,
      "observation": "line2 matches=[]"
    },
    {
      "satisfied": true,
      "observation": "line3 matches=[]"
    },
    {
      "satisfied": true,
      "observation": "line4 matches=[]"
    },
    {
      "satisfied": true,
      "observation": "line5 matches=[]"
    },
    {
      "satisfied": true,
      "observation": "line6 matches=[]"
    }
  ]
}
```


The official tests of this trial pass, in agreement with the revised check.

Acceptance by counterexamples. In the trial of Appendix D.1, the first version of the check stored the quoted requirement in a variable. The framework's source validation requires the quotation to appear literally in each `expect`, returned the error below, and the verifier revised the check into the version shown there.

Framework → verifier: validation error on the first version


```
{
  "error": "expect needs a literal requirement= public-task quote"
}
```


Before the revised check could act, the framework ran it on the three counterexamples of its claim. It returned FAIL on each, and returned the FAIL of Appendix D.1 on the current state, a usable verdict. A separate model review of these outputs then judged that the check measures its claim:

Framework: acceptance test on the counterexamples


```
{"counterexample": {"operation": "empty", "origin": "model", "path": "/usr/local/bin/povray"}, "
    ↪ status": "fail", "observations": ["named file /usr/local/bin/povray exists and is
    ↪ nonempty: False"]}
{"counterexample": {"operation": "delete", "origin": "standard", "path": "/usr/local/bin/povray
    ↪ "}, "status": "fail", "observations": ["named file /usr/local/bin/povray exists and is
    ↪ nonempty: False"]}
{"counterexample": {"operation": "empty", "origin": "standard", "path": "/usr/local/bin/povray"},
    ↪  "status": "fail", "observations": ["named file /usr/local/bin/povray exists and is
    ↪ nonempty: False"]}
{"accepted": true}
```


Verifier: acceptance review


```
{
  "aligned": true,
  "reason": "The check executes the specified sanity-check command, captures the executable\
    ↪ u2019s exit status, and checks its output for numerical rendering statistics. These
    ↪ directly measure the claim rather than file or implementation proxies. The observed exit
    ↪ status of 1 and missing shapes.inc demonstrate a genuine failure to complete the required
    ↪  sanity render."
}
```