BOXBOX: RECURSIVE HARNESS SELF-IMPROVEMENT FOR LONG-RUNNING PERFORMANCE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated harness optimization improves agents by learning from trajectories and task feedback. Existing offline methods use reference answers to optimize harnesses for reuse on held-out tasks. However, many real-world tasks lack reference solutions, while long runs expose new difficulties that offline-optimized harnesses may not address. To study harness optimization in these settings, we introduce PERFRACE, a benchmark for long-running performance optimization of software projects without reference solutions, and further investigate whether agents can use task-specific experience to recursively improve their own harnesses during an ongoing task. To this end, we introduce BOXBOX, which enables agent-triggered harness revision and kernel-managed hot reloading while preserving execution context. We evaluate BOXBOX on PERFRACE under matched test-time budgets, comparing it with Frozen Harness, offline Harness Evolution, and Harness Scaling. BOXBOX achieves higher speedup with more efficient use of task time and model cost, reaching a $\mathrm{Speedup@4h}$ of $1.70\times$, compared with $1.54\times$ for the strongest baseline. Code is available at `https://anonymous.4open.science/r/BoxBox-1C35/`.

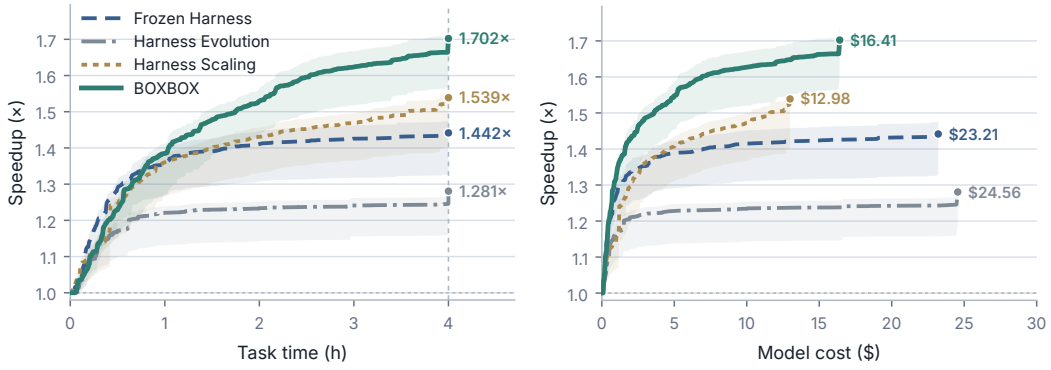


Figure 1: Performance of BOXBOX and baselines on 50 PERFRACE test tasks, with three runs per method and task. Curves show aggregate best-so-far correct speedup against task time (left) and mean cumulative model cost per run (right), sampled at matching task times, with shaded bands indicating 95% fixed-task run-resampling intervals. BOXBOX achieves higher speedup with greater time and cost efficiency than the baselines.

1 INTRODUCTION

Large language model (LLM) agents interact with their environment through harness that determines both the available actions and the observations they receive (Yang et al., 2024; Merrill et al., 2026; Lin et al., 2026). Recent studies have shown that harness design can substantially affect agent performance and behavior, even when the underlying model is fixed (Lin et al., 2026; Wang et al., 2026b). These findings have motivated automatic optimization of agent harness, in which an optimizer analyzes agent trajectories and evaluation feedback, proposes modifications to the harness, and evaluates the resulting agent on benchmark tasks.

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.

Existing offline harness optimization methods (Park et al., 2026; Zhang et al., 2026b; Lin et al., 2026; Wang et al., 2026b) often use reference answers or task-completion feedback to diagnose failures, and revise harnesses to improve performance on held-out tasks. This setup aligns with existing agent benchmarks that evaluate whether an agent completes a specified task (Jimenez et al., 2024; Merrill et al., 2026; Froger et al., 2026).

However, many real-world tasks are open-ended and long-running: they lack reference solutions or a clearly defined point of completion and allow continued progress, making end-state evaluations alone insufficient to capture how agents progress throughout these runs. In this setting, applying offline harness optimization raises two challenges. First, the lack of reference solutions makes it difficult to identify useful harness revisions from execution experience: prior evaluations show that harness evolution does not reliably improve performance without external correctness feedback (Wang et al., 2026b). Second, continued execution exposes new task-specific difficulties that harness revisions learned from other tasks may not address (Nie et al., 2026; Wang et al., 2026b).

To study harness optimization in open-ended and long-running settings, we introduce PERFRACE, a benchmark for optimizing performance of real-world software projects without reference solutions. Our experiments show that offline Harness Evolution (HE) underperforms Frozen Harness on PERFRACE (Figure 1), with no consistent improvement during training or validation (Appendix A.3), consistent with prior findings of unreliable gains and limited generalization (Wang et al., 2026b).

Nevertheless, the current task's trajectory can reveal difficulties encountered during execution and provide task-specific evidence for harness revision (Nie et al., 2026). We therefore investigate whether an agent can use this experience to improve its own harness during the ongoing task, making harness optimization an online, task-specific process. Existing methods allocate budget between task execution and harness optimization through an outer loop, with harness updates scheduled by predefined triggers or a separate meta-agent (Wang et al., 2026b; Liu et al., 2026; Zhang et al., 2026c). However, external control limits the task-solving agent's ability to respond to emerging difficulties. First, despite observing task progress, the agent cannot decide whether to spend its remaining budget on task execution or harness revision. Second, updates occur between task attempts or execution segments, delaying their use in the ongoing task.

To address these limitations, we introduce BOXBOX, which supports agent-triggered revision of the full executable harness through kernel-managed hot reloading. When requested, the kernel pauses task execution and launches an agent review session to diagnose difficulties from the trajectory and revise the harness. After validation, it resumes the task with preserved execution context under the revised harness, which governs both subsequent execution and future reviews, enabling recursive harness improvement. We evaluate BOXBOX on PERFRACE against Frozen Harness, offline Harness Evolution and externally controlled Harness Scaling under matched task budgets, comparing best-so-far correct speedup and model cost. BOXBOX achieves a $\mathrm{Speedup@4h}$ of $1.70\times$, compared with $1.44\times$ for Frozen Harness, $1.28\times$ for Harness Evolution, and $1.54\times$ for Harness Scaling.

Our main contributions are summarized as follows:

- We introduce PERFRACE, a benchmark for evaluating agents on open-ended performance optimization tasks without reference solutions, tracking correctness-preserving improvements to real-world software throughout task execution.
- We present BOXBOX, which combines agent-triggered, full-harness revision from task-specific experience with kernel-managed hot reloading of the runtime while preserving execution context.
- Our experiments show that BOXBOX achieves higher best-so-far correct speedup with greater time and cost efficiency than the evaluated baselines, supporting harness optimization as part of an agent's own capabilities.

2 RELATED WORK

Automated harness optimization. Prior work has extensively studied the automated optimization of prompts for LLM-based systems (Zhou et al., 2023; Pryzant et al., 2023; Yuksekgonul et al., 2025; Khattab et al., 2024; Agrawal et al., 2025). Subsequent work extends automated optimization to agents, using execution trajectories and task feedback to improve their memory, skills, and workflows (Zhao et al., 2024; Zhang et al., 2025; Ni et al., 2026; Hu et al., 2024; Zhang et al., 2024).

Meta-Harness and AHE further extend automated optimization to executable harness code, using trajectories and evaluation feedback to guide iterative revisions (Lee et al., 2026; Lin et al., 2026).

Building on this setting, offline methods revise harnesses using reference answers or task-completion feedback from training tasks, aiming to improve held-out performance (Park et al., 2026; Zhang et al., 2026b; Lin et al., 2026). However, an evaluation finds unreliable gains without correctness feedback and limited transfer from reusable harnesses, but reports gains from Harness Scaling, which alternates task attempts and harness revisions at test time (Wang et al., 2026b).

Following this task-specific direction, JIT-Agent synthesizes a tailored harness for each task before execution begins (Zhang et al., 2026a). TTHE further explores harness optimization without reference answers, using trajectories and execution-derived feedback to revise executable harnesses on unlabeled test batches (Nie et al., 2026). Continual Harness moves harness optimization into ongoing task execution through an outer loop that periodically invokes a refiner while preserving the task state (Karten et al., 2026). In BOXBOX, the task-solving agent decides when to allocate its remaining budget to full-harness revision, while a separate kernel manages runtime hot reloading and preserves execution context.

Long-Horizon and Open-Ended Agent Evaluation. Existing agent benchmarks cover diverse capabilities but commonly evaluate tasks with explicit end states, reporting task success, pass rates, or final artifact quality (Mialon et al., 2023; Liu et al., 2024b; Jimenez et al., 2024; Xie et al., 2024; Merrill et al., 2026; Sun et al., 2026). Recent iterative optimization benchmarks evaluate solution quality but still use reference implementations, human performance, or reference outputs for task construction or scoring (Chan et al., 2025; Lyu et al., 2026; Imajuku et al., 2025; Xu et al., 2026; Mang et al., 2025; Zhu et al., 2026). Closer to our setting, Frontier-Eng and FrontierSWE support iterative optimization of engineering designs and software projects, with tasks curated from established problems and expert contributions (Chi et al., 2026; Chu et al., 2026).

In contrast, PERFRACE constructs long-running performance optimization tasks from real-world software projects and workloads without requiring reference optimizations. Its construction pipeline provides a scalable way to continually source new tasks from public software projects.

Benchmarks for Performance Optimization. Shypula et al. (2024) introduce PIE for evaluating LLMs' ability to make performance-improving edits to single-file competitive programming solutions while preserving functional correctness. Subsequent benchmarks broaden the optimization domains to GPU kernels, general-purpose numerical programs, and algorithmic redesign, while retaining function- or file-level optimization tasks (Du et al., 2024; Huang et al., 2024; Liu et al., 2024a; Waghjale et al., 2024; Ouyang et al., 2025; Press et al., 2025).

GSO and SWE-Perf extend evaluation to repository-level performance optimization, deriving tasks and human reference optimizations from GitHub commits and pull requests (Shetty et al., 2025; He et al., 2026). SWE-fficiency and FormulaCode further incorporate a targeted workload per task and multiple community-maintained workloads, respectively (Ma et al., 2026; Sehgal et al., 2026).

These repository-level benchmarks construct tasks around a targeted performance test, whereas real performance engineering starts from an end-to-end workload and discovers bottlenecks through profiling. PERFRACE adopts this workload-first formulation, leaving agents to identify and implement correctness-preserving improvements without a known target patch.

3 PERFRACE

PERFRACE benchmarks real-world software performance optimization with three design principles: (1) *Live*: projects are continually sourced from public GitHub repositories; (2) *Open-ended*: tasks use project workloads without reference optimizations or known target patches; and (3) *Long horizon*: agents iteratively optimize code and choose when to submit for feedback within a fixed budget.

3.1 TASK FORMULATION

Each PERFRACE task provides the agent with an editable project workspace, a fixed workload that the agent can run locally, and access to an external Judge. For official evaluation, the Judge uses a separate copy of this workload and additional held-out correctness cases.

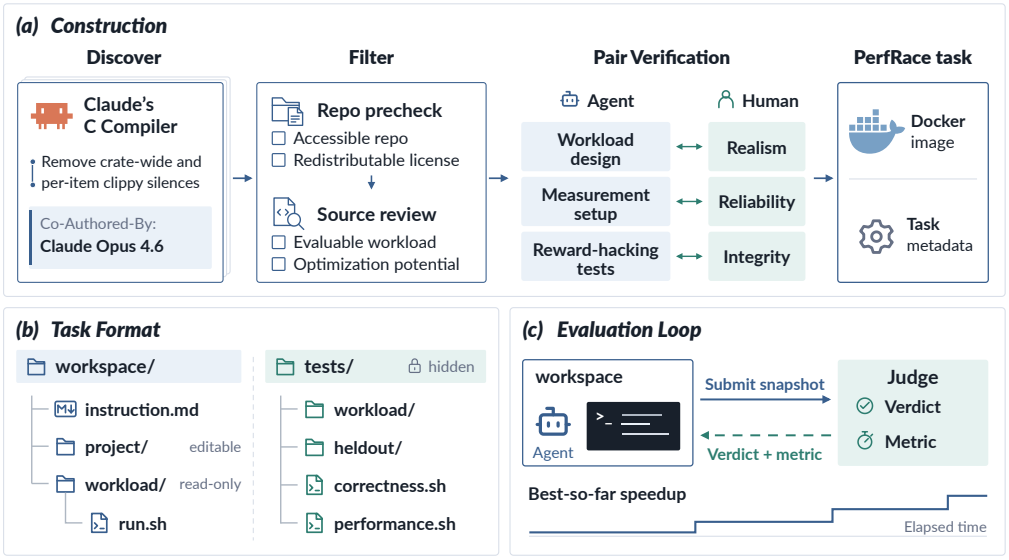


Figure 2: Overview of PERFRACE. (a) Construction discovers projects from GitHub coding-agent activity, applies a two-stage Filter comprising a deterministic precheck and source-based assessment, and uses Agent–Human Pair Verification to produce a PERFRACE task. (b) Task format separates the agent workspace from hidden evaluation files. (c) The evaluation loop repeatedly submits workspace snapshots to the Judge and tracks best-so-far correct speedup.

During the evaluation loop, the agent iteratively modifies the project and submits changes to the Judge for feedback. Let t_i denote the evaluation start time of submission i and s_i its score, with $s_i = 0$ for incorrect submissions. Scores are attributed retrospectively to evaluation start times; verification time also counts toward the task budget. We include the unmodified project as a baseline submission with $t_0 = 0$ and $s_0 = 1$. We define the best-so-far correct speedup as $S(t) = \max_{i:t_i \le t} s_i$, which the agent aims to maximize within a fixed task budget B.

3.2 CONSTRUCTION

We construct PERFRACE by identifying real projects with optimization potential and building reliable tasks around realistic workloads (Figure 2(a)). Each PERFRACE task adopts the Harbor Task Format (Merrill et al., 2026), as illustrated in Figure 2(b).

Discovery. To identify less-explored optimization opportunities beyond mature, heavily maintained repositories, we search GitHub coding-agent activity for candidate projects. Specifically, we identified 103,475 candidate projects from public GitHub pull requests and branch pushes made by known coding agents between May and August 2026 (Appendix B.3).

Filter. We filter candidate projects in two stages. First, deterministic prechecks verify repository and source-revision accessibility, license redistributability, and repository size and popularity, retaining 7,257 candidates. We then use GPT-5.6-Luna to assess each candidate's setup process, evaluable workload, runtime feasibility, and optimization potential from its source code. Of these candidates, 239 pass this assessment and proceed to Pair Verification (Appendix B.3).

Pair Verification. We use agent–human Pair Verification to construct runnable tasks and assess their quality against three criteria (Appendix B.4):

- **Realism.** The workload and metric represent meaningful project use.
- **Reliability.** Repeated equivalent runs support the same correctness and performance conclusions.
- **Integrity.** The score reflects genuine task completion rather than an invalid or shortcut solution.

Specifically, we first let the agent derive workload inputs and scale from real project use and propose correctness checks, measurement boundaries, and performance metrics for human review. Second,

the agent configures warmup and runs repeated tests to detect flakiness and distinguish genuine speedups from noise; the reviewer uses these results to approve or revise the measurement protocol. Finally, the agent conducts red-team tests against reward hacking using invalid, incomplete, hard-coded, and work-skipping solutions; the reviewer requests verifier fixes as needed before approving the task for packaging and release. Eventually, we curated 100 tasks, each requiring an average of 9.1 hours of curation, 12 agent–human interactions, and $11.35 in agent cost; detailed statistics are provided in Appendix B.3.

3.3 Benchmark Composition

The resulting benchmark contains 100 tasks across seven domains (Figure 3): *developer tools* (30 tasks) compiles, builds and validates software; *data systems* (25 tasks) ingests, indexes and queries data; *scientific computing* (15 tasks) runs simulations and numerical solvers; *media processing* (9 tasks) renders graphics and audio; *geometry* (8 tasks) processes maps and 3D models; *machine learning* (7 tasks) trains models and runs inference; and *security* (6 tasks) encrypts data and analyzes attack logs. Each task pairs a real project with a fixed workload; for example, one task optimizes CCC while it compiles SQLite. Appendix B.2 provides a detailed characterization of the tasks.

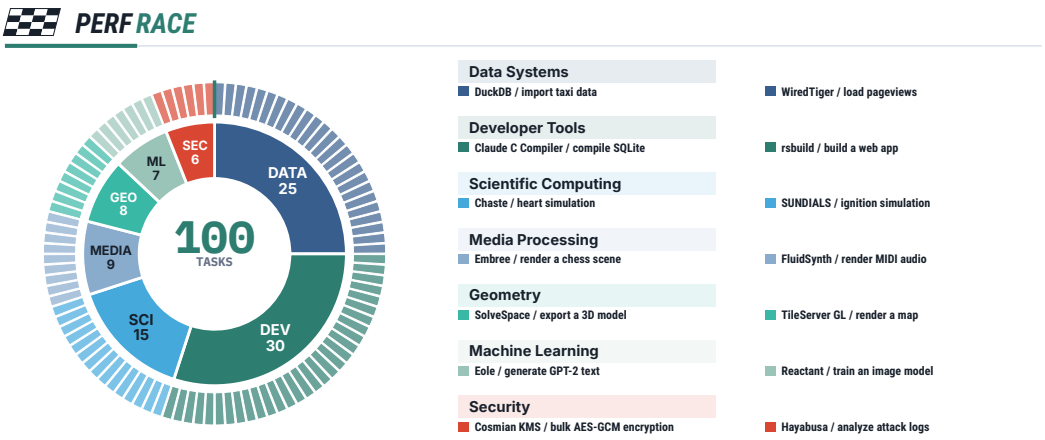


Figure 3: Composition of PerfRace: seven workload domains (inner ring), 100 tasks (outer ring), and representative project–workload pairs (right).

3.4 Evaluation

In the evaluation loop, the agent chooses when to submit snapshots for correctness and performance feedback from the Judge (Figure 2(c)). The score s_i is the speedup over the unmodified project for correct submissions and zero otherwise. Runs without a correctness-preserving improvement receive a score of $1\times$. During the final five minutes, the agent is prompted to end optimization and leave the project in a coherent state for the Judge's final evaluation.

Prior studies show that agents can inflate scores by tampering with evaluation code or accessing reference outputs (Wang et al., 2026a; Von Arx et al., 2025). We therefore use *shadow verification*, evaluating project snapshots in separate containers while keeping verifier code and held-out test data out of the agent's workspace. Time spent in shadow verification counts toward B. Appendix B.5 provides further details on shadow verification.

4 BoxBox

We now present BoxBox, which uses a kernel to support agent-triggered updates on demand and seamless task resumption through hot reloading, allowing the agent to recursively improve its harness from task-specific experience.

4.1 Problem Formulation

Consider a fixed language model $\mathcal{M}$ and an executable harness H. Let e_k and τ_k denote the environment state and accumulated trajectory at interaction step k, respectively (Yao et al., 2023). Under a fixed H, one interaction step consists of

$$\begin{aligned} a_k &\sim p_{\mathcal{M},H}(\cdot \mid \tau_k), \qquad a_k \in \mathcal{A}, \\ (e_{k+1}, o_{k+1}) &\leftarrow \mathrm{Execute}_H(e_k, a_k), \\ \tau_{k+1} &= \tau_k \oplus (a_k, o_{k+1}). \end{aligned}$$

Here, a_k is a tool call from the action space $\mathcal{A}$ exposed by H, and $p_{\mathcal{M},H}$ is its distribution induced by H constructing model inputs from τ_k and invoking $\mathcal{M}$. $\mathrm{Execute}_H$ resolves and executes a_k through the harness's tool implementation, updates the environment state, and processes the resulting feedback into o_{k+1}; $\oplus$ appends the action–observation pair to the history.

We allow the harness to be revised from H to H' during the ongoing task using the available interaction history, with the following effect on subsequent interactions:

$$H \longrightarrow H' \quad \Longrightarrow \quad \big(p_{\mathcal{M},H}, \mathrm{Execute}_H\big) \longrightarrow \big(p_{\mathcal{M},H'}, \mathrm{Execute}_{H'}\big). \tag{1}$$

4.2 Kernel-Managed Harness Updates

BoxBox exposes the full executable harness for task-specific revision, including tool implementations, control flow, and state management. Let $\mathcal{H}$ denote the set of harness implementations compatible with the execution interface. Previous work limits revisions to predefined components of H, restricting H' to $\mathcal{H}_{\mathrm{bounded}}(H) \subset \mathcal{H}$ (Zhang et al., 2026b; Lin et al., 2026; Park et al., 2026). In contrast, we allow revisions $H' \in \mathcal{H}$, letting the agent choose which components to modify and how extensively based on task-specific experience.

In BoxBox, the harness is the editable agent implementation, while the runtime is its running version that controls the agent's execution and maintains its state. Updating the harness while the agent is running changes the code controlling its execution, meaning that a faulty update can interrupt the task and leave the agent unable to recover on its own. To address this risk, BoxBox introduces a separate kernel that manages runtime hot reloading and can safely retain or restore the previous runtime if an update fails validation or activation.

Specifically, we track harness versions as Git commits so the kernel can select the revision to load. On the agent side, we register `boxbox()` as a tool that yields control to the kernel, extending the action space to $\mathcal{A}' = \mathcal{A} \cup \{\texttt{boxbox()}\}$. When the agent invokes `boxbox()`, the kernel suspends task execution and preserves the agent's execution context, including the accumulated trajectory, terminal session, and workspace state. Next, the kernel launches a review session for the agent to examine the trajectory, revise the harness as needed, and commit any changes. After the review session concludes, the kernel validates any new commits, loads the revised runtime on success, and otherwise retains the current runtime. During activation, the kernel transfers the preserved execution context to the new runtime and reattaches the existing terminal session. On successful activation, subsequent interactions use $p_{\mathcal{M},H'}$ and $\mathrm{Execute}_{H'}$, as per Eqn. 1. Finally, the kernel returns control to the agent, allowing it to resume the ongoing task.

Beyond agent-initiated updates, runtime faults also trigger kernel-managed recovery and review based on the recorded fault evidence. Appendix C provides further implementation details.

4.3 Agent-Controlled Harness Optimization

Building on this mechanism, the agent triggers harness reviews on demand based on task progress and the remaining budget. Unlike prior methods that use an outer loop (Lee et al., 2026; Karten et al., 2026; Liu et al., 2026), we let the task-solving agent use free-form reasoning to allocate budget B between project-code optimization and harness revision to maximize $S(B)$.

Within the review session, we provide a default skill (Appendix D.3) that guides the agent through a simple self-review strategy based on diagnosis and harness revision:

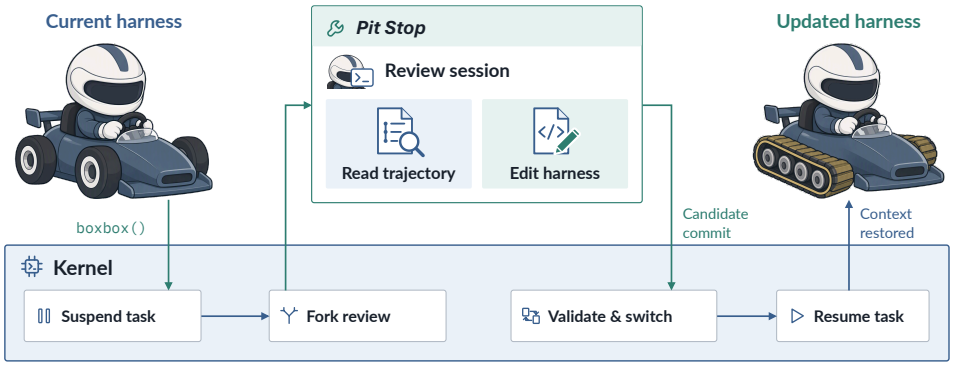


Figure 4: **BoxBox mechanism.** The agent requests a review on demand, triggering a trap that transfers control to the kernel. The kernel then launches the review, validates revisions, and hot-reloads the runtime with preserved execution context and rollback if activation fails.

- **Diagnosis.** The agent examines the trajectory for execution failures, repeated work, and successful patterns that could be automated or reused. It uses these observations to identify limitations of the current harness, such as missing capabilities or costly tool interactions.
- **Revision.** The agent selects harness revisions based on their expected benefit within the remaining budget, such as adding guidance, improving tools, or changing execution logic. For example, repeated tool calls can be combined into a reusable tool.

In subsequent reviews, the agent can inspect Git history alongside execution records to assess previous harness changes and decide whether further revisions are needed.

5 Experiments

5.1 Experimental Setup

Baselines. We compare BoxBox with three baselines: **Frozen Harness** (Frozen) retains the initial harness throughout task execution. **Harness Evolution** (HE) optimizes the harness offline on training tasks and selects it by validation performance, then holds it fixed during test-task execution. **Harness Scaling** (HS) uses an external controller to revise the harness from each completed attempt's feedback and start a new agent session, preserving project-code progress but resetting the execution context. Both HE and HS are adapted from Wang et al. (2026b). Following Meta-Harness (Lee et al., 2026), all methods start from the same initial harness H_0, Terminus-KIRA (KRAFTON AI & Ludo Robotics, 2026), a simple terminal-based harness that executes shell commands and returns their outputs to the model.

Evaluation protocol. We split the 100 PerfRace tasks into 40 training and 10 validation tasks for HE, and 50 held-out test tasks for evaluating all methods. We use GPT-5.6 Luna with high reasoning effort for both task execution and harness optimization. We evaluate each method with three independent runs per test task, each sharing a four-hour budget ($B = 4\text{h}$) across task execution, harness optimization, and evaluation. Detailed configurations are provided in Appendix A.

Metrics. To account for randomness, we summarize task i's best-so-far correct speedup across $R = 3$ runs as $\bar{S}_i = \left(\prod_{r=1}^{R} S_i^{(r)}(B)\right)^{1/R}$. Let $\overline{\text{Cost}}_i$ be its mean model cost in US dollars, including harness optimization. Across $N = 50$ tasks, we use harmonic mean speedup to limit the influence of extreme values (Shetty et al., 2025) and arithmetic mean task-level speedup per dollar:

$$\text{Speedup@4h} = \frac{N}{\sum_{i=1}^{N} \bar{S}_i^{-1}}, \qquad \text{Speedup/\$} = \frac{1}{N} \sum_{i=1}^{N} \frac{\bar{S}_i}{\overline{\text{Cost}}_i}. \tag{2}$$

Reported mean model cost averages $\overline{\text{Cost}}_i$ across tasks. Cost accounting and statistical details appear in Appendix A.

Table 1: Results on 50 test tasks with three four-hour runs per task, reporting $\mathrm{Speedup@4h}$, mean model cost per run, and mean task-level speedup per dollar.

Method	$\mathrm{Speedup@4h}$	Mean model cost	Speedup/$ ↑
FROZEN	1.442	$23.22	0.2142
HE	1.281	$24.56	0.1243
HS	1.539	$12.98	0.2872
BOXBOX	**1.702**	$16.43	**0.4886**

5.2 MAIN RESULTS

We report performance and model cost in Table 1 and plot speedup against task time and cumulative model cost in Figure 1. Appendix A.5 provides optimization trajectories for all test tasks.

Offline harness optimization does not reliably improve performance. HE reaches $1.281\times$ at four hours, below FROZEN (Table 1). Without reference solutions, training shows no consistent improvement, while training and validation gains fail to transfer to long-running test tasks. Appendix A.3 provides detailed results and failure analysis. Including at least $1,376.46 in offline training and validation costs raises HE's mean cost per test run from $24.56 to at least $33.73.

Task-specific harness optimization supports continued improvement. Both HS and BOXBOX improve over FROZEN, reaching $1.539\times$ and $1.702\times$ at four hours, respectively, compared with $1.442\times$ for FROZEN (Table 1). Both also reduce mean model cost, to $12.98 and $16.43 per run, respectively, compared with $23.22 for FROZEN. Figure 1 shows the corresponding speedup trajectories against elapsed task time and mean cumulative model cost. Over the second half of the four-hour budget, the FROZEN curve largely levels off, increasing from $1.410\times$ to $1.442\times$, while HS improves from $1.431\times$ to $1.539\times$ and BOXBOX from $1.530\times$ to $1.702\times$. These trends show that task-specific harness optimization supports further progress as execution continues.

Agent-controlled harness optimization improves performance and efficiency. Compared with HS, BOXBOX reaches higher speedups at the same elapsed task time and at the same cumulative model cost, demonstrating more efficient optimization throughout the task (Figure 1). HS's outer loop separates budget allocation from the task-solving agent's decisions and defers harness updates until an attempt ends, delaying harness revisions and their benefits until the next attempt. Specifically, each update starts a new agent session, reducing total model cost but requiring context reconstruction that can lead to repeated exploration. In contrast, BOXBOX lets the agent decide when to spend its remaining budget on harness revision and resume the ongoing task with its execution context preserved. Appendix A.4 provides a detailed analysis. These results show that BOXBOX improves task performance while making more efficient use of task time and model cost.

5.3 ANALYSIS AND ABLATION STUDIES

To assess the effectiveness of agent-controlled harness optimization, we analyze how BOXBOX revises its harness and ablate review timing and editable scope.

Harness Optimization Analysis.

Review frequency. BOXBOX invokes a median of five reviews per run, with 92% of runs containing three to six (Figure 5(a)). Review therefore recurs throughout task execution, providing repeated opportunities to revise the harness from task-specific experience.

Budget allocation. Review sessions occupy a median of 2.1% of the task budget and 13.8% of model cost per run (Figure 5(b)), leaving most time for task execution. Despite this additional expenditure, BOXBOX achieves higher speedup at lower total model cost than FROZEN.

Modification scope. Feedback and prompt rules are the most frequent modifications, while updates also affect skills, tools, execution logic, and fault handling (Figure 5(c)). Cases in Appendix A.6 show how these changes add capabilities, automate waiting, and revise context management.

Ablation Studies.

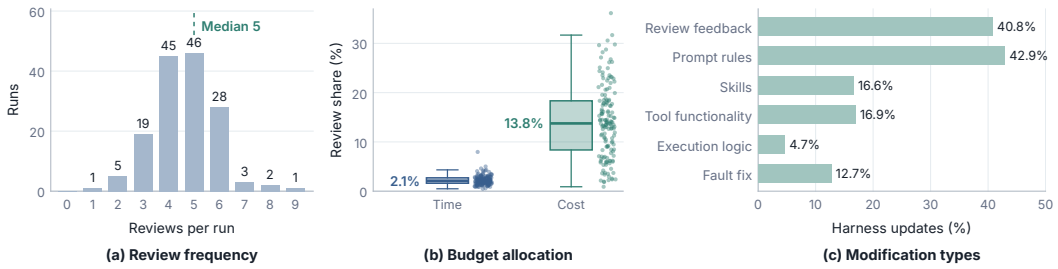


Figure 5: **Harness optimization analysis.** (a) Number of reviews per run across 150 test runs, including agent-triggered and fault-recovery reviews. (b) Per-run review time as a share of the four-hour task budget and review cost as a share of total model cost within that budget; dots represent runs, and boxes show the median and interquartile range. (c) Modification types across a separate collection of 620 applied, nonempty harness updates from BOXBOX runs.

Table 2: Results on 14 test tasks with three runs per task and a four-hour budget. Model cost includes task execution and harness review; Speedup/$ averages task-level ratios.

Condition	Trigger	Editable scope	Speedup@4h	Mean model cost	Speedup/$
Periodic BOXBOX	Every 30 min	Full	1.508	$21.58	0.1338
Bounded BOXBOX	Agent-initiated	Predefined scope	1.349	$16.07	0.1132
Full BOXBOX	Agent-initiated	Full	**1.584**	$15.07	**0.3452**

Periodic triggering. Following the scheduled refinement in Continual Harness (Karten et al., 2026), Periodic BOXBOX schedules reviews every 30 minutes with full-harness editing. It achieves lower speedup at higher model cost than agent-triggered BOXBOX (Table 2). Scheduled reviews can interrupt ongoing code optimization, whereas agent-controlled triggering preserves continuity between proposing, implementing, and evaluating code changes. Scheduled reviews also consume model calls even when they conclude that no harness revision is needed.

Bounded editing. Following Self-Harness (Zhang et al., 2026b), Bounded BOXBOX retains agent-controlled triggering but restricts each review to a predefined scope. It achieves lower speedup than full BOXBOX (Table 2). Restricting each review to a single surface may favor local corrections to the current problem over reconsidering the broader optimization process. Full-harness editing instead lets the agent determine the scope of revision from task-specific experience, leaving room for broader changes as the search develops.

Together with the main results, the harness optimization analysis and ablations support harness optimization as part of the task-solving agent's own capabilities: the agent initiates harness reviews while keeping most of its time for the task, and ablations show that agent-controlled triggering and full-harness editing contribute to better performance.

6 CONCLUSION

In this work, we introduced PERFRACE, a benchmark for long-running performance optimization of real-world software without reference optimizations, and BOXBOX, which combines agent-triggered, full-harness revision from task-specific experience with kernel-managed hot reloading that preserves execution context. On PERFRACE, offline Harness Evolution yields inconsistent gains, whereas BOXBOX achieves higher best-so-far correct speedup and greater time and cost efficiency than the evaluated baselines. It achieves a Speedup@4h of $1.70\times$, compared with $1.54\times$ for the strongest baseline, and reduces mean model cost by 29.3% relative to Frozen Harness. These results support harness optimization as an agent capability for improving how it solves a task while continuing its execution.

AI USE STATEMENT

The authors determined the research design, mathematical formulation, experimental methodology, and interpretation of results. In addition to the LLM agents studied in this work, we used generative AI tools to write implementation and data-analysis code and assist with benchmark construction. We also used these tools for literature search, language polishing, and figure production, including AI-generated illustrative assets. Benchmark tasks underwent the agent–human verification process described in Section 3.2. The authors take responsibility for the paper's content, including its claims, results, code, and other artifacts.

ETHICS STATEMENT

PERFRACE is constructed from publicly available software projects and workload inputs. Curation includes checking the applicable modification and redistribution terms, with human review of workload provenance and permission to redistribute inputs (Appendix B). The task catalog records source projects and licenses to acknowledge upstream contributions and support responsible reuse.

REPRODUCIBILITY STATEMENT

The repository for code and experimental artifacts is available at `https://anonymous.4open.science/r/BoxBox-1C35/`, with the following contents:

- **Method implementations:** BOXBOX, Frozen Harness, Harness Evolution, Harness Scaling, and the ablation variants.
- **Benchmark curation pipeline:** project discovery, automated filtering, and agent–human pair verification.
- **Tasks and evaluation:** task assets, environment setup, and correctness and performance evaluation utilities.
- **Experiment configurations:** model settings, task budgets, and instructions for running the methods.
- **Experimental artifacts:** harness revisions, training and validation results, and task-level performance and cost records.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, 2025.

Yizhe Chi, Deyao Hong, Dapeng Jiang, Tianwei Luo, Kaisen Yang, Boshi Zhang, Zhe Cao, Xiaoyan Fan, Bingxiang He, Han Hao, et al. Frontier-Eng: Benchmarking self-evolving agents on real-world engineering tasks with generative optimization. *arXiv preprint arXiv:2604.12290*, 2026.

Evan Chu, Rajan Agarwal, Abishek Thangamuthu, Brendan Graham, and Justus Mattern. FrontierSWE: Benchmarking coding agents at the limits of human abilities. `https://www.frontierswe.com/blog/v1`, 2026.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 16601–16622, 2024. doi: 10.52202/079017-0529.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. GAIA2: Benchmarking LLM agents on dynamic and asynchronous environments. *arXiv preprint arXiv:2602.11964*, 2026.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. SWE-Perf: Can language models optimize code performance on real-world repositories? In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Seoul, South Korea, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. EffiBench: Benchmarking the efficiency of automatically generated code. In *Advances in Neural Information Processing Systems*, volume 37, pp. 11506–11544, 2024. doi: 10.52202/079017-0367.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A benchmark for long-horizon objective-driven algorithm engineering. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

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*, 2024.

Seth Karten, Joel Zhang, Tersoo Upaa Jr., Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, et al. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, volume 2024, pp. 54928–54958, 2024.

KRAFTON AI and Ludo Robotics. Terminus-KIRA: Boosting frontier model performance on Terminal-Bench with minimal harness, 2026. URL `https://github.com/krafton-ai/kira`.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*, 2024a.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. EvoX: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, 2024b.

Bohan Lyu, Yucheng Yang, Siqiao Huang, Jiaru Zhang, Qixin Xu, Xinghan Li, Xinyang Han, Yicheng Zhang, Huaqing Zhang, Runhan Huang, et al. MLS-Bench: A holistic and rigorous assessment of AI systems on building better AI. *arXiv preprint arXiv:2605.08678*, 2026.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Seoul, South Korea, 2026.

Qiuyang Mang, Wenhao Chai, Zhifei Li, Huanzhi Mao, Shang Zhou, Alexander Du, Hanchen Li, Shu Liu, Edwin Chen, Yichuan Wang, et al. FrontierCS: Evolving challenges for evolving intelligence. *arXiv preprint arXiv:2512.15699*, 2025.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. *arXiv preprint arXiv:2311.12983*, 2023.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. TTHE: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel K. Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, Nathanael Bosch, Eli Meril, Albert Steppi, Arman Zharmagambetov, Fangzhao Zhang, David Pérez-Piñeiro, Alberto Mercurio, Ni Zhan, Talor Abramovich, Kilian Lieret, Hanlin Zhang, Shirley Huang, Matthias Bethge, and Ofir Press.

AlgoTune: Can language models speed up general-purpose numerical programs? In *Advances in Neural Information Processing Systems*, volume 38, pp. 118982–119027, 2025. doi: 10.52202/085713-3585.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, 2023.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. FormulaCode: Evaluating agentic optimization on large codebases. In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Seoul, South Korea, 2026.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, Chris Duroiu, Haoyun Zhang, Jeffrey Lin, et al. Agents' last exam. *arXiv preprint arXiv:2606.05405*, 2026.

Sydney Von Arx, Lawrence Chan, and Beth Barnes. Recent frontier models are reward hacking. METR Blog, June 2025. URL https://metr.org/blog/2025-06-05-recent-reward-hacking/.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15362–15376. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.859.

Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? systematically auditing AI agent benchmarks with BenchJack. *arXiv preprint arXiv:2605.12673*, 2026a.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. *arXiv preprint arXiv:2607.12227*, 2026b.

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. *arXiv preprint arXiv:2404.07972*, 2024.

Zhangchen Xu, Junda Chen, Yue Huang, Dongfu Jiang, Jiefeng Chen, Hang Hua, Zijian Wu, Zheyuan Liu, Zexue He, Lichi Li, Shizhe Diao, Jiaxin Pei, Jinsung Yoon, Hao Zhang, Mengdi Wang, Radha Poovendran, Misha Sra, Alex Pentland, and Zichen Chen. AutoLab: Can frontier models solve long-horizon auto research and engineering tasks? *arXiv preprint arXiv:2606.05080*, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025.

Guibin Zhang, Leo Lu, Fangzhou Xie, Kang Zhu, Junhao Wang, Zhifei Xie, Zhaochen Yu, Zihang Liu, Zhongxiang Sun, Qiankun Li, Yue Liao, Heng Chang, Xiaobin Hu, Qibing Ren, Wangchunshu Zhou, Chuanrui Hu, Yafeng Deng, and Shuicheng Yan. JIT-Agent: Scaling harness intelligence via just-in-time harness evolution. *arXiv preprint arXiv:2608.25593*, 2026a.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026b.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024.

Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, Bang Liu, Chenglin Wu, and Yuyu Luo. Harnessing agentic evolution. *arXiv preprint arXiv:2605.13821*, 2026c.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *International Conference on Learning Representations*, 2023.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, et al. EdgeBench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026.

A Experimental Details

We describe the experimental environment and evaluation details in Appendix A.1, followed by the experiments for Frozen Harness, Harness Evolution, and Harness Scaling in Appendices A.2–A.4. Appendix A.5 provides additional per-task optimization trajectories, and Appendix A.6 examines how BoxBox harness revisions support task execution.

A.1 Experimental Environment and Evaluation Details

Experimental environment. We conduct our experiments on an Azure Standard_D96ads_v5 virtual machine with 96 vCPUs backed by an AMD EPYC 7763 processor and 378 GiB of system memory, running Ubuntu 24.04 with Docker.

Time and cost accounting. Figure 1 and Table 1 use the same 600 test runs: 50 tasks, four methods, and three runs per method and task. For all methods, elapsed task time sums recorded agent-execution and verification phases. Agent execution includes harness review; environment setup is excluded. Only evaluations started within the four-hour budget contribute to $\mathrm{Speedup@4h}$. We calculate model cost from recorded token usage using the rates in Table 3. Reported mean model costs per test run include all recorded model calls; cost and token curves report cumulative usage at each task-budget time point.

Table 3: Model token rates used for cost calculation (USD per million tokens).

Model	Uncached input	Cached input	Output
GPT-5.6-Luna	\$1.00	\$0.10	\$6.00
GPT-5.6-Terra	\$2.50	\$0.25	\$15.00

Statistical analysis. The bands in Figure 1 show the 2.5th and 97.5th percentiles of 5,000 resampled benchmark runs over the fixed 50-task set. Each sample independently selects one of the three observed runs per task and aggregates the selected scores using the harmonic mean. The central curves instead use all three runs per task, with the geometric-then-harmonic aggregation defined in Equation 2. In the cost panel, speedup intervals are plotted at the mean cumulative cost for each task-time point; they do not represent uncertainty in cost.

A.2 Frozen Harness

We instantiate the initial harness H_0 with Terminus-KIRA, whose agent loop executes shell commands and returns their outputs to the model, as described in Section 5.1. The harness remains unchanged throughout each run while the agent iteratively optimizes the project code. We evaluate it on the same 50 test tasks as the other methods, with three independent four-hour runs per task.

A.3 Harness Evolution

Experimental configuration. Table 4 summarizes the HE pipeline and its recorded model costs. Training revisits the same 40 tasks in three reshuffled epochs; iteration i evaluates h$i-1$ on five tasks and produces hi. Validation compares h8, h16, and h24 with fresh baseline measurements.

Table 4: HE configuration and recorded model costs. Training and validation costs include retries and reruns and are lower bounds where usage records are missing.

Stage	Tasks	Evaluated runs	Budget per run	Cost (\$)
Training	40	24 iterations $\times$ 5 tasks	45 min	200.34
Validation	10	3 harnesses $\times$ 3 runs/task	2 h	1,176.12
Test	50	h24, 3 runs/task	4 h	3,683.34

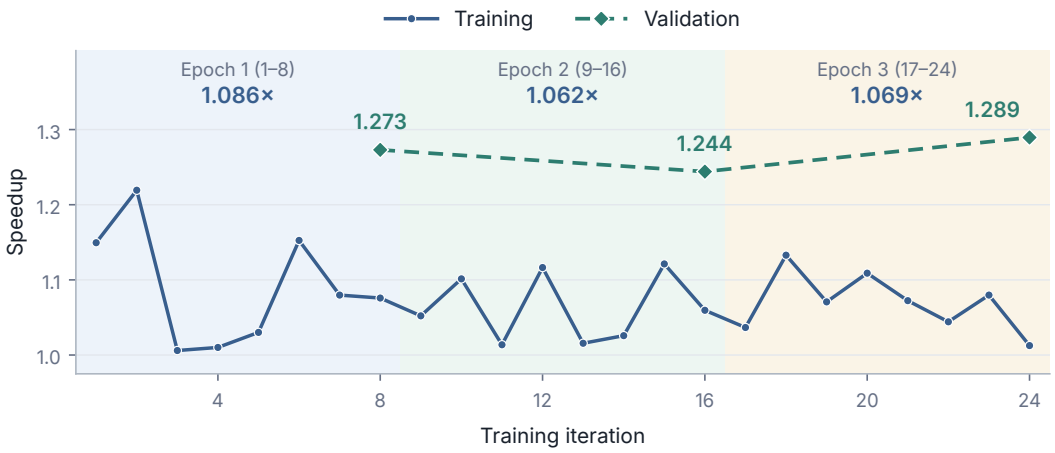


Figure 6: HE training and validation performance. Each training point aggregates five task runs under h$i-1$; shaded epochs cover three visits to the same 40 tasks, with their aggregate speedups shown above. Validation points evaluate the resulting h8, h16, and h24 on ten tasks with three runs each. Scores use the aggregation in Equation 2, with one run per training task. Training and validation have different tasks and budgets.

Training, validation, and test results. Training performance does not improve consistently across epochs (Figure 6); among the 16 batches revisiting previously evaluated tasks, seven improve and nine regress. Validation performance also fluctuates, with h24 achieving the highest score among the evaluated checkpoints. We therefore use h24 for testing, where it achieves $1.281\times$, below Frozen Harness at $1.442\times$. These results show that additional offline revisions do not reliably improve training performance or yield a test-time advantage, consistent with prior findings of limited transfer (Wang et al., 2026b). The h24 test evaluation followed inspection of h16 test results; h24 itself had already been frozen. Including the recorded offline cost raises mean cost to at least $33.73 per test run.

Failure analysis. Without reference solutions, HE must infer potential failure patterns from trajectories and evaluation outcomes to decide how to revise the harness. Before iteration 5's revision, its diagnostic report linked selected measurements in `mrst-norne-import` to unchanged code and in `mssql-extension-taxi-copy` to reverted code, and identified others without a traceable candidate version. The optimizer interpreted these observations as a failure to connect measured gains to code changes and introduced a mandatory evidence check before evaluation (Figure 7). This turned a diagnosis drawn from training trajectories into a rule applied to subsequent tasks.

Iteration 5: require local evidence before evaluation.

```
+ workdir = environment.task_env_config.workdir
+ record = await environment.exec(
+     "cat -- .autoresearch/promotion.json",
+     cwd=workdir,
```

Iteration 22: reuse the resolved repository root.

```
+ self._research_workdir = workdir
  ...
- workdir = environment.task_env_config.workdir
+ workdir = self._research_workdir or environment.task_env_config.workdir
```

Figure 7: HE introduces an evidence check at iteration 5 and revises its repository-root handling at iteration 22. The fallback to the configured workspace remains in h24.

In one `mirage-kubernetes-triage` test run, this check exposed a repository-layout assumption that the offline revisions had not resolved. The agent established Git metadata and validated its evidence inside the editable repository, but completion checks continued to inspect the parent workspace. The trajectory records 473 blocked completion requests over approximately 3 hours and 46 minutes, with repeated attempts to satisfy the check. The intended safeguard thus became an obstacle: repairing the evidence inside the project did not change where the harness looked for it.

This case connects the two challenges in Section 1: without reference solutions, revision directions must be inferred from observed behavior and outcomes; during long-running execution, the resulting rules encounter task-specific conditions not resolved by offline training. Here, further interaction revealed the mismatch but could not correct the fixed harness, turning a local incompatibility into repeated lost opportunities for optimization.

A.4 Harness Scaling

Following the Harness Scaling adaptation described in Section 5.1, an external controller reviews feedback from each completed attempt and revises the harness before starting a new agent session. The project workspace and its optimization progress persist across attempts, but the previous session's context is not restored. Task execution, harness revision, and evaluation share the four-hour budget. We evaluate this procedure on the same 50 test tasks, with three independent runs per task.

Cost efficiency and session continuity. HS has a lower mean model cost than BoxBox ($12.98 versus $16.43 per run), but also achieves less speedup ($1.539\times$ versus $1.702\times$). Its mean task-level speedup per dollar is also lower: 0.2872, compared with 0.4886 for BoxBox (Table 1). BoxBox has a higher task-level ratio on 31 of the 50 tasks, with a median of 0.1576 versus 0.1176 for HS. Arithmetic averaging remains sensitive to large ratios: the five highest-ratio tasks contribute 65.6% of the sum for BoxBox. Figure 1 provides a complementary comparison, showing higher speedups for BoxBox at matched cumulative costs.

Analysis. *Harness updates wait until an attempt ends.* The controller records show repeated alternation between task execution and harness revision. The available round summaries record 2,132 attempts, with a median of 12 per run. The median interval between one attempt ending and the next beginning is 3.37 minutes across 1,983 boundaries. These intervals include external harness optimization and runtime transition; they are not measurements of avoidable delay. The controller begins harness revision after the preceding attempt ends, so difficulties encountered within an attempt cannot immediately change the harness governing that attempt. This separates the task-solving agent's assessment of progress from the decision to allocate time to harness revision. BoxBox instead exposes that decision to the agent within ongoing execution.

Session restarts require rebuilding context. HS retains project-code progress but starts the next attempt without the preceding session's context. The new session must reconstruct relevant observations and reasoning from the workspace and revised harness. For example, in one `eucalypt-eval-speed` run, successive terminal records begin by locating the project and inspecting the workload script; the second and third attempts also repeat a shallow listing of project files. This is direct evidence of renewed orientation after a restart, although such checks can be useful and are not by themselves evidence of wasted exploration. If earlier conclusions or rejected hypotheses are not recorded in the retained artifacts, the same reconstruction process can also lead to repeated experiments or revisited alternatives. BoxBox preserves execution context across updates, allowing the agent to continue using those observations without reconstructing them in a new session. The records support this difference in continuity, but do not isolate the amount of time or cost lost to redundant exploration.

A.5 Per-Task Optimization Trajectories

Figures 8–12 show optimization trajectories for all 50 test tasks. Each curve geometrically averages three runs, following the scoring and time-accounting conventions of Figure 1. The vertical scale varies by task.

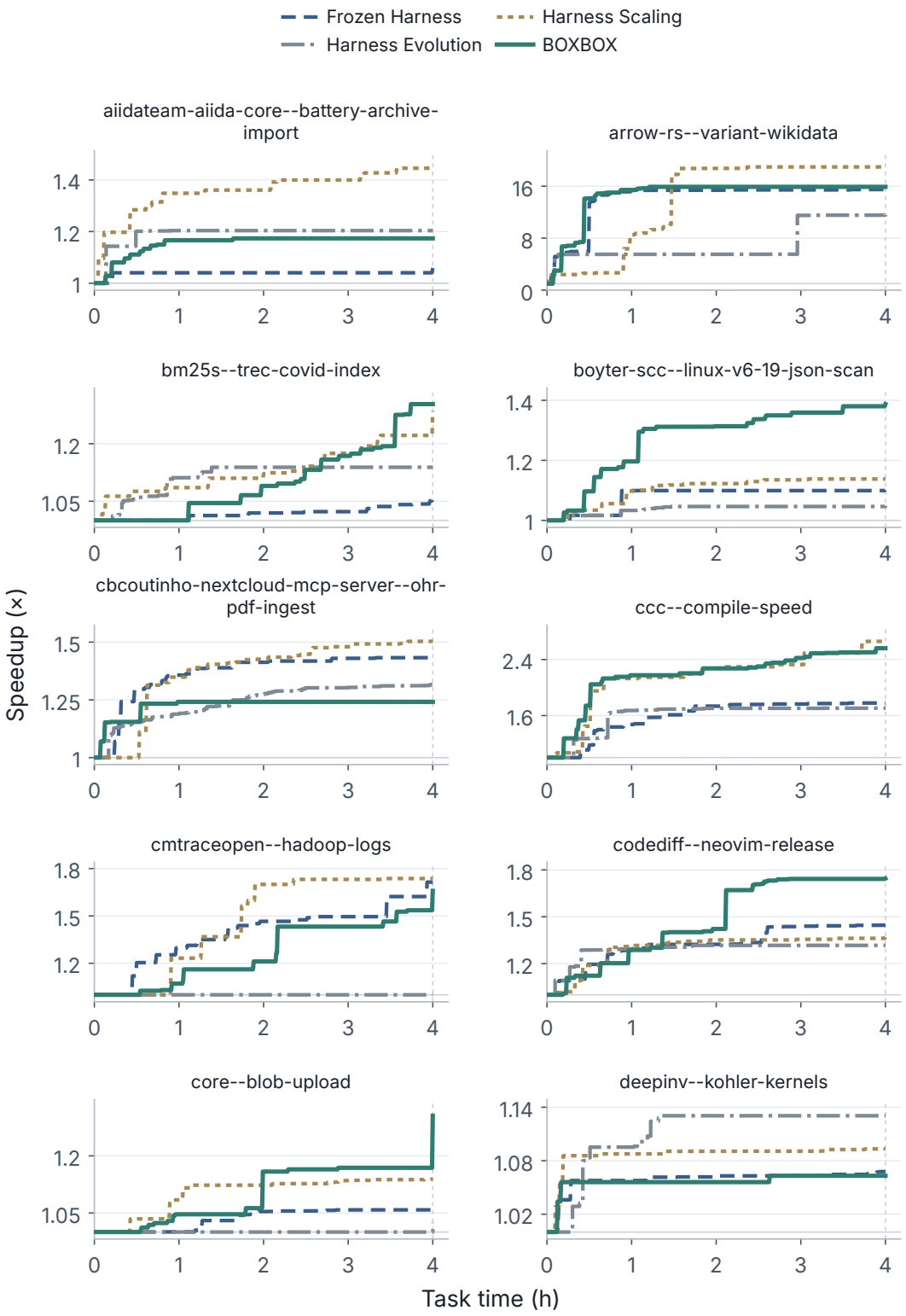


Figure 8: Per-task optimization trajectories (1/5).

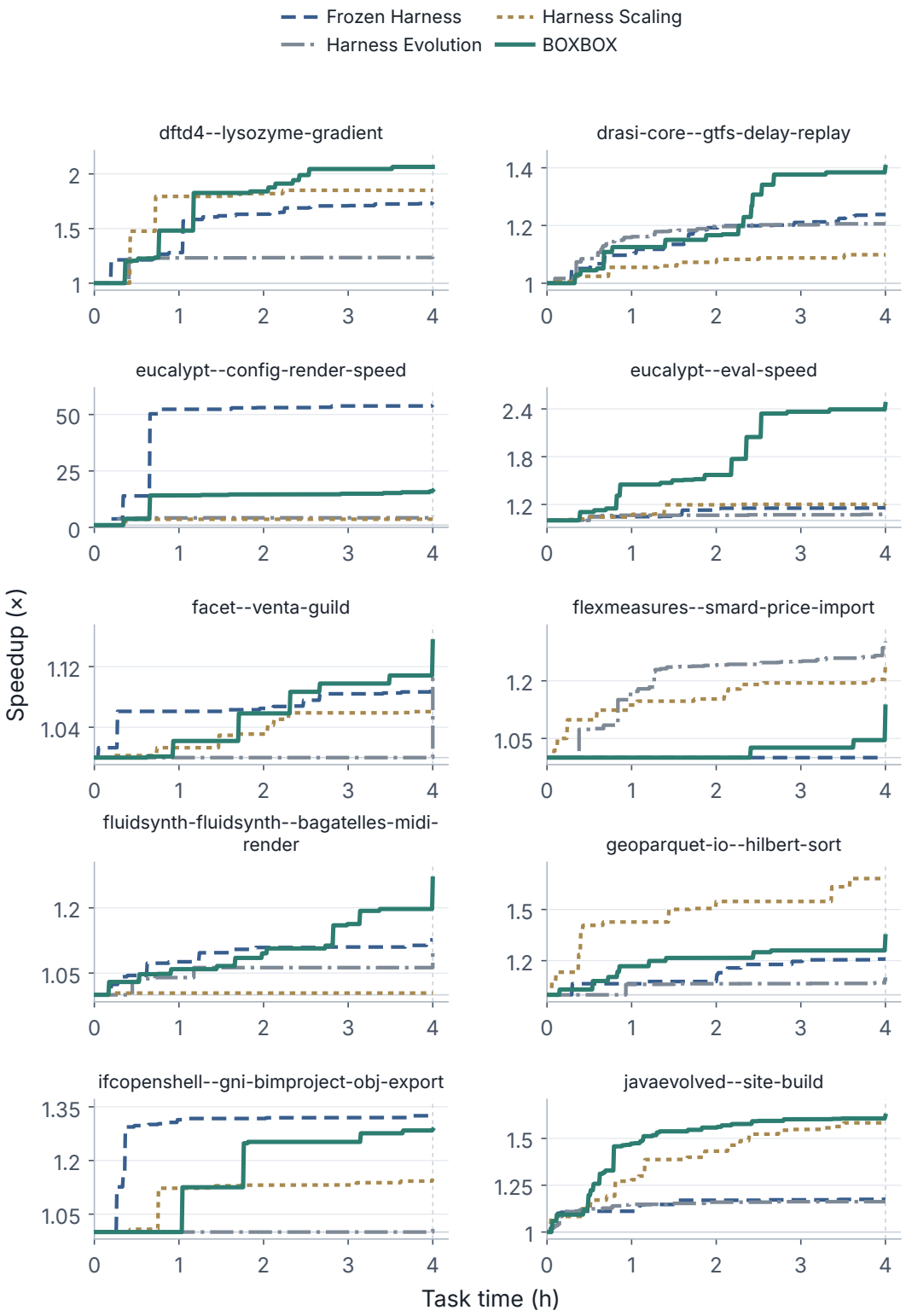


Figure 9: Per-task optimization trajectories (2/5).

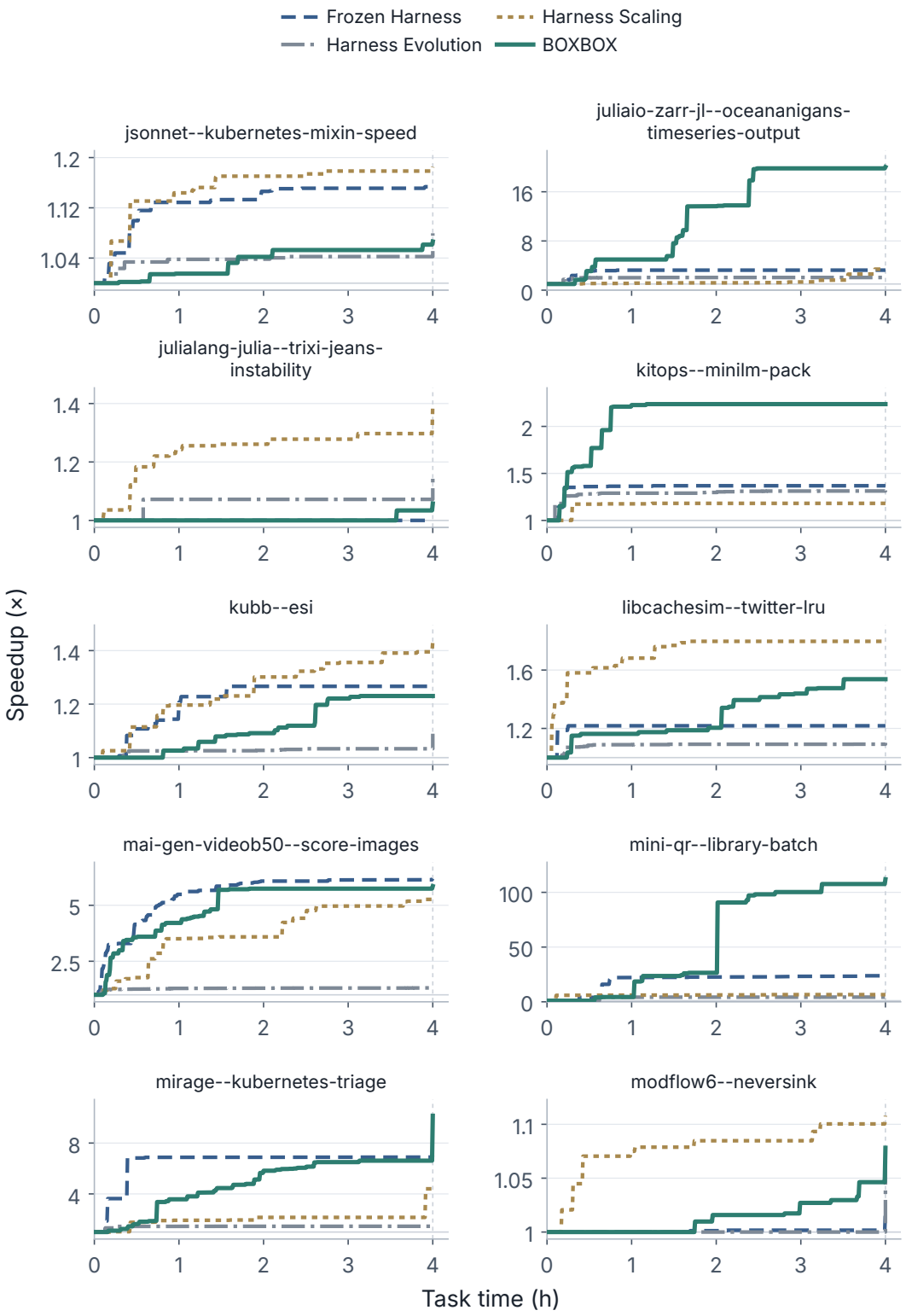


Figure 10: Per-task optimization trajectories (3/5).

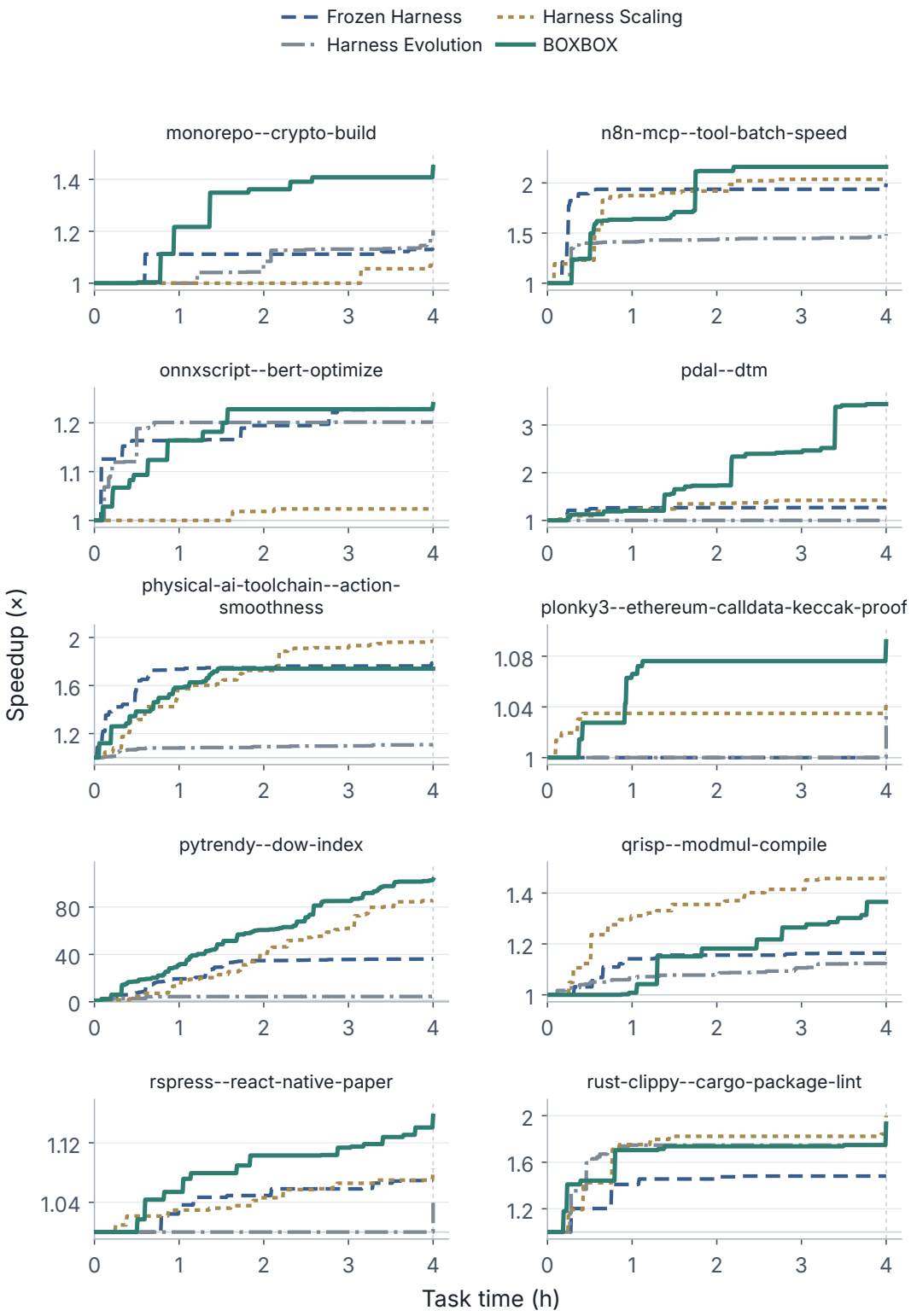


Figure 11: Per-task optimization trajectories (4/5).

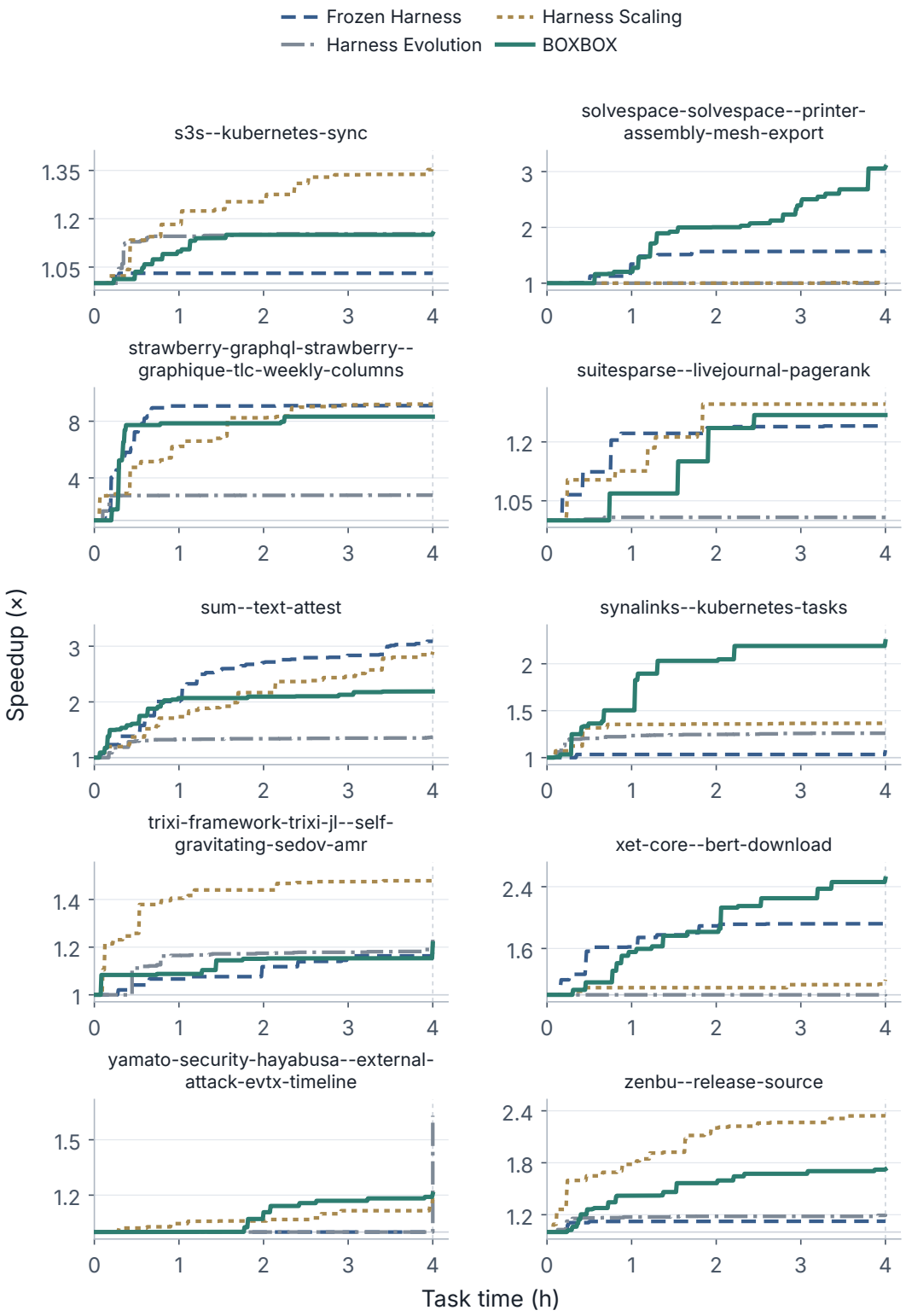


Figure 12: Per-task optimization trajectories (5/5).

A.6 CASE STUDY

To explain how BOXBOX supports effective task execution, we examine seven harness updates that change agent guidance, experimental tools, or execution behavior. These cases complement the aggregate results by showing how task-specific harness revisions address concrete difficulties encountered during optimization. While Figure 5(c) reports the prevalence of modification categories, Table 5 illustrates what those modifications look like in individual tasks.

Table 5: Representative updates for the six modification categories in Figure 5(c). Case numbers refer to the detailed analyses below; task IDs follow the catalog in Table 7.

Case	Category	Task ID	Modification
A.6.1	Review feedback	`n8n-mcp--tool-batch-speed`	Redirect the next experiment toward SQLJS conversion and statement overhead.
A.6.2	Prompt rules	`sum--text-attest`	Require alternating A/B measurements when apparent gains are close to noise.
A.6.3	Skills	`eucalypt--eval-speed`	Require new evidence before retrying a discarded hypothesis.
A.6.4	Tool functionality	`n8n-mcp--tool-batch-speed`	Add a tool for paired workload measurements and response comparison.
A.6.5		`rust-clippy--cargo-package-lint`	Extend internal command polling to reduce model-directed waiting turns.
A.6.6	Execution logic	`trixi-framework-trixi-jl--self-gravitating-sedov-amr`	Add a message-count-based context-compaction path.
A.6.7	Fault fix	`rust-clippy--cargo-package-lint`	Replace an exception on shell exit with a session-recovery call.

The following cases proceed from changes in agent guidance to changes in executable behavior. Cases A.6.1–A.6.3 examine review feedback, prompt rules, and skills that guide experiment selection and evaluation. Cases A.6.4–A.6.5 examine tool implementations for measurement and waiting. Cases A.6.6–A.6.7 turn to context management and terminal recovery within the agent loop. Together, they illustrate how full-harness editing can affect both the direction of the search and the process used to carry it out.

A.6.1 REVIEW FEEDBACK: REDIRECTING AN EXPERIMENT

Task: `n8n-mcp--tool-batch-speed`

Search context. The review describes an existing accepted candidate and a logging experiment that did not improve local timing. It identifies another profiler clue: repeated row/object conversion and statement management in the SQLJS adapter. The update records a concrete next direction in the review-feedback handoff, without changing the tool implementation or the agent loop.

```
+ The remaining profiler clue is the SQLJS adapter's repeated `all/get`
+ row/object conversions and statement lifecycle overhead. Treat this as
+ a hypothesis, not a conclusion: inspect the exact hot path and design
+ one general correctness-preserving experiment ...
+
+ ... Use a separate branch from main, build, run the exact workload with
+ paired repeated clean-vs-experiment medians, and compare response hashes.
+ Reject changes within noise or with any response mismatch.
```

What changes. The feedback connects a specific observation to an experiment, a comparison procedure, and a rejection criterion. It asks the continuing agent to preserve the current candidate while testing whether a different source of overhead is removable. The full handoff also discourages revisiting the logging experiment and submitting unchanged or speculative candidates. Thus, the

change concerns how the existing capabilities should be used next, rather than adding a missing capability.

Implication for optimization. The intended benefit is a better allocation of experiments: avoid repeating a rejected direction and investigate a distinct cost instead. This can be useful even when the current harness already supports profiling, editing, and testing. The example also separates a useful review from a code rewrite: a review can supply task-specific guidance while leaving executable harness behavior intact. The earlier accepted improvements mentioned in the feedback precede this update and are not evidence of its effect.

A.6.2 Prompt rules: comparing noisy measurements

Task: `sum--text-attest`

Decision problem. When a candidate's apparent improvement is similar to measurement noise, one fast run provides a weak basis for deciding whether to keep it. The patch adds an explicit comparison procedure to the main prompt. It does not add a new measurement tool or automatically run an experiment.

```
+ When a candidate's apparent gain is comparable to run-to-run noise, do not
+ infer from one fast first run. Keep the unchanged `main` reference and
+ candidate branch, warm both in the same controlled procedure, then run
+ alternating fresh-process A/B repetitions with identical output checks
+ and compare medians (and spread) before accepting or rejecting the hypothesis.
+ Align this local procedure with the official workload boundary before
+ using a tiny timing difference to justify submission.
```

What changes. The added rule specifies which versions to compare, how to prepare them, how to order repeated measurements, and which evidence to use in the decision. Keeping the reference unchanged prevents the comparison target from drifting during the experiment. Identical output checks keep timing interpretation tied to the candidate's behavior, while medians and spread provide a more informative summary than a selected fast sample.

Implication for optimization. This intervention spends more measurement effort on ambiguous candidates in exchange for a better-supported acceptance decision. It is therefore not inherently a cost-saving patch: extra runs consume time. Its intended value is to reduce decisions based on unstable timing evidence and the downstream work those decisions can induce. Actual compliance still depends on the agent following the prompt.

A.6.3 Skills: using discarded experiments

Task: `eucalypt--eval-speed`

Search context. The existing autoresearch skill is extended with a rule for revisiting rejected hypotheses. Instead of treating a prior rejection as either irrelevant or permanent, the update specifies what new evidence would justify reopening it.

```
+ Treat a `status=discarded` research note as a negative prior. Reopen a
+ discarded hypothesis only when new evidence changes the causal model: the
+ current `main` changed the affected operation count, profiling now shows a new
+ per-operation cost, the prior patch was confounded by an implementation error,
+ or the benchmark path changed. Before creating the retry branch, write down
    the
+ specific changed assumption and its expected removable end-to-end cost; if the
+ only reason is that time has passed or other optimizations made the program
+ faster, choose a fresh profile-led hypothesis instead.
```

What changes. The skill now links hypothesis selection to the research history retained in Git. A retry needs an explicit changed assumption, such as a new operation count or evidence that the

previous implementation was faulty. Merely having time remaining is not a sufficient reason. Conversely, an earlier failure does not rule out a direction whose relevant conditions have genuinely changed.

Implication for optimization. The patch aims to avoid repeated investigation of an unchanged hypothesis while preserving the ability to revisit it as the project evolves. Unlike the n8n-mcp feedback, which selects a concrete next direction, this rule can guide multiple future decisions. Its distinction from the preceding prompt example is the carrier: it extends a reusable skill rather than the main prompt template. Neither intervention enforces the decision through executable control logic.

A.6.4 TOOL FUNCTIONALITY: PAIRED MEASUREMENT

Task: `n8n-mcp--tool-batch-speed`

The two tool examples address different sources of repeated work. Paired measurement standardizes how experimental evidence is collected, while command polling changes how often waiting requires a model decision.

Preceding behavior. The existing shell tool could be used to assemble comparisons. The update introduces `benchmark_workload`, registers it as a new tool, adds argument parsing and dispatch, and implements a helper that generates a complete baseline–candidate comparison script.

```
+ "name": "benchmark_workload",
+ ...
+ "properties": {
+     "baseline_ref": {"type": "string"},
+     "candidate_ref": {"type": "string"},
+     "repetitions": {"type": "integer", "minimum": 1, "maximum": 3},
+ },
+ "required": ["baseline_ref", "candidate_ref"],
```

What changes. The agent supplies two committed Git references and a bounded repetition count. The helper checks the target layout and source-tree state, records the original reference, and backs up the runtime database. For each pair it checks out and builds the baseline and candidate, runs the same workload, records elapsed time, and compares their response files. A nonzero workload exit or a response mismatch terminates the comparison instead of silently producing a timing summary.

```
+ for pair in $(seq 1 __REPETITIONS__); do
+   run_one __BASELINE__ baseline "$pair" || exit $?
+   run_one __CANDIDATE__ candidate "$pair" || exit $?
+   if cmp -s "/tmp/n8n-mcp-bench-baseline-${pair}.jsonl" \
+             "/tmp/n8n-mcp-bench-candidate-${pair}.jsonl"; then
+     printf 'pair=%s responses=identical\n' "$pair"
+   else
+     printf 'pair=%s responses=DIFFERENT\n' "$pair"
+     exit 14
+   fi
+ done
```

The code accumulates timing samples and reports a median for each version. The workload timing interval starts after the build, so build time is excluded from the reported workload duration but still consumes task budget. An exit handler attempts to restore the original Git reference and database. The ordering is baseline then candidate within each pair, not a randomized schedule.

Why this is more than a prompt edit. The preceding sum example requests a procedure; this update implements one. Given a valid call, version selection, execution order, error handling, response comparison, and summary generation are performed by code. The agent still chooses which candidate to test and how to act on the result, but need not regenerate the comparison procedure on every occasion. This is task-specific functionality: the implementation refers to the n8n-mcp repository, database, and workload output.

Implication for resource use. Packaging the procedure can reduce repeated command construction and intermediate model decisions, while making comparisons more consistent. It does not eliminate the builds or workload repetitions. A well-constructed single shell script could already perform these operations; the new tool makes that procedure reusable and parameterized. Its terminal dispatch uses a bounded observation window, so a long comparison may still require later waiting calls. The separate polling example addresses that overhead. Local response equality is an experimental check, not a replacement for the official verifier.

A.6.5 TOOL FUNCTIONALITY: REDUCING MODEL CALLS FOR COMMAND POLLING

Task: `rust-clippy--cargo-package-lint`

Preceding behavior. In the Rust Clippy harness, a command is submitted to a persistent terminal and its completion marker is polled for the requested observation duration. If the command is still running when that duration expires, control returns to the model. The model can then issue an empty-keystroke command to continue waiting. A long build can consequently generate repeated model turns whose only decision is to wait again.

What changes. The patch extends the polling budget inside the command runner. For ordinary requested durations of at most 60 seconds, the new budget is 180 seconds; the implementation also caps it at the existing block timeout. The completion-marker check still permits an early return. This changes the maximum observation window, not the execution speed of the underlying build.

```
+ AUTO_COMMAND_WAIT_SEC = 180
  ...
+ wait_budget = min(
+     BLOCK_TIMEOUT_SEC,
+     max(command.duration_sec, AUTO_COMMAND_WAIT_SEC),
+ )
  ...
- while time.monotonic() - start < command.duration_sec:
+ while time.monotonic() - start < wait_budget:
```

The tool description is updated alongside the implementation. It tells the agent to rely on the internal wait and request an additional wait only when the command outlasts the new window.

```
- "It is always possible to wait again if the prior output has not finished, "
- "by running empty keystrokes with a duration on subsequent requests to wait
    longer. "
+ "The harness automatically continues waiting when the completion marker has "
+ "not appeared, up to 180 seconds per tool call, and returns a progress
    snapshot "
+ "at that bound. "
+ "Use an empty-keystroke wait only when a command exceeds that bound; "
+ "use C-c to interrupt foreground work. "
```

Where model cost is avoided. An internal terminal poll reads command state without asking the language model to generate a new action. In contrast, returning an unfinished observation and asking the model what to do next can incur input-token charges and output-token charges for another wait action. The next input may contain a large continuing context even when the useful new observation is only that the build is still running. Cached input can lower that charge under the experiment's pricing schedule, but does not make the whole request costless.

For a fixed command episode, let $\mathcal{W}$ denote the model calls made solely to request another wait. The model expenditure attributable to those calls is

$$C_{\text{wait}} = \sum_{j \in \mathcal{W}} \left(r_u I_j^u + r_c I_j^c + r_o O_j \right),$$

where I_j^u, I_j^c, and O_j are uncached input, cached input, and output token counts, and the r terms are their respective per-token rates. If the longer internal window removes some of these calls,

their charges form the direct saving opportunity. Fewer wait messages can also reduce subsequent context growth. Terminal polling still uses execution resources, and the build still consumes wall-clock budget.

Benefit and trade-off. The mechanism is most useful when a command finishes within the extended window and no intermediate decision is needed. For example, a command finishing after the initial short window but before 180 seconds can return its completed output without a model-generated continuation wait. The trade-off is delayed model access to intermediate output: a stalled or failing command may deserve earlier inspection or interruption. Commands exceeding the extended window can still require further interaction. Net savings depend on the actual avoided calls, context caching, and later behavior; the diff alone does not establish a dollar saving. The patch concerns terminal-command polling, not polling the status of an LLM API response.

A.6.6 Execution logic: context compaction

Task: `trixi-framework-trixi-jl--self-gravitating-sedov-amr`

Preceding behavior. The loop already has an optional proactive summarization path controlled by `_enable_summarize`. The update adds another path before that check, so the new trigger can operate even when the inherited optional path is disabled or has not reached its own threshold.

```
+ if (
+     original_instruction
+     and len(chat.messages) >= 64
+     and len(chat.messages) >= self._last_forced_compaction_messages + 24
+ ):
+     self._last_forced_compaction_messages = len(chat.messages)
+     proactive_summary_result = await self._force_proactive_compaction(
+         chat, original_instruction, self._session,
+     )
+     if proactive_summary_result:
+         prompt, subagent_refs = proactive_summary_result
+         self._pending_subagent_refs = subagent_refs
+         self._pending_handoff_prompt = prompt
  if original_instruction and self._enable_summarize:
      ...
```

What changes. The condition uses the current message count: at least 64 messages and at least 24 more than the count recorded at the last attempt. It records the count before calling a new helper, which invokes the existing summarizer under a timeout and returns `None` on an exception. Successful results become the pending handoff prompt and subagent references. The counter measures messages, not tokens or elapsed time; the patch changes when the loop attempts compaction rather than asking the model to select a new tool.

Implication for cost and continuity. Compaction can replace repeated transmission of a long interaction history with a shorter continuation context. Unlike command polling, which can remove model calls, this intervention primarily targets their input size and incurs the cost of producing a summary. Net benefit depends on the summary length, the number of future turns, and cache behavior. Summarization can also omit useful detail. The failure handler provides a path to continue with the existing history if the attempt fails; it does not guarantee that every summary preserves all relevant information. Thus, the example illustrates editable context-management logic, not an unconditional cost reduction.

A.6.7 Fault fix: terminal recovery

Task: `rust-clippy--cargo-package-lint`

Preceding behavior and update. At the beginning of an agent-loop iteration, the previous implementation checks whether the persistent shell is alive and raises an exception if it has exited. A fault-triggered review replaces that branch with a call to the existing session-management helper.

```
- if not await self._with_block_timeout(self._session.is_session_alive()):
-     raise RuntimeError("The persistent terminal shell exited")
+ # The previous episode may have outlived the persistent shell.
+ # Restart it for the next model turn without replaying any
+ # command whose completion was uncertain.
+ await self._with_block_timeout(self.ensure_agent_session())
```

Implication for continuing the task. The change moves terminal-liveness handling from an immediate exception to an existing ensure-session path. Avoiding automatic replay matters because a command may have modified files before its completion became uncertain; repeating it could duplicate effects. However, starting a replacement shell is not equivalent to reconstructing its previous process state. Shell-local variables and in-flight commands may require inspection before work can safely continue. The diff therefore shows a targeted recovery policy, while successful task resumption must be established from subsequent execution. It does not add a new optimization tool or change the kernel's recovery implementation.

B PERFRACE: BENCHMARK DETAILS

The PERFRACE appendix is organized as follows: Section B.1 compares PERFRACE with related benchmarks; Section B.2 characterizes its 100 tasks; Section B.3 reports construction and curation statistics; Section B.4 describes the Pair Verification protocol and quality control; and Section B.5 details the evaluation protocol and analyzes gains from longer optimization budgets.

B.1 BENCHMARK CURATION AND EVALUATION

Table 6 compares PERFRACE with representative ultra-long-horizon agent benchmarks and code-optimization benchmarks along task type, task construction, and evaluation procedure. Existing ultra-long-horizon benchmarks largely assemble fixed suites of hand-crafted or expert-curated tasks. In contrast, PERFRACE combines live GitHub discovery with agent–human Pair Verification to continually construct validated optimization tasks from real-world projects and workloads. Compared with existing code-optimization benchmarks, it does not derive tasks from known performance patches and evaluates progress repeatedly rather than only at task completion.

A workload specifies the inputs and operations used to exercise a project. A microbenchmark measures a narrowly scoped part of that work, such as a selected API call or operation. For example, SWE-fficiency includes a performance workload that repeatedly times a NumPy `einsum` call (Ma et al., 2026). An end-to-end workload instead measures a complete project-use workflow. PERFRACE adopts this broader measurement boundary, leaving agents to profile the workflow and identify which components to optimize. The distinction is therefore the scope of the measured computation: targeted operations versus an end-to-end workflow, rather than whether the benchmark supplies a workload.

Table 6: Comparison with related benchmarks. Microbenchmarks measure targeted operations (for example, selected API calls), whereas end-to-end workloads measure complete project-use workflows.

Benchmark	Task type	Task construction	Evaluation
PERFRACE	Repository optimization on end-to-end workloads	Live GitHub discovery followed by agent–human Pair Verification	Iterative correctness and performance feedback from an external Judge
AutoLab (Xu et al., 2026)	Diverse metric-driven optimization tasks	Expert-curated optimization tasks	Local evaluation during the task; held-out final evaluation
EdgeBench (Zhu et al., 2026)	Cross-domain iterative problem-solving tasks	Expert-curated day-scale tasks	Local inner loop; repeated submission-gated Judge feedback
FrontierSWE (Chu et al., 2026)	Ultra-long-horizon software engineering and research tasks	Real-world problems curated with academic and industry experts	Final graded submission after up to 20 hours
Terminal-Bench (Merrill et al., 2026)	General terminal tasks	Crowdsourced tasks with multi-stage human review	Final-only test-based evaluation
GSO (Shetty et al., 2025)	Microbenchmark-based repository optimization	Repository commit histories and generated performance tests	Final-only microbenchmark-based evaluation
SWE-Perf (He et al., 2026)	Microbenchmark-based repository optimization	Historical performance PRs and selected repository tests	Final-only microbenchmark-based evaluation
SWE-fficiency (Ma et al., 2026)	Microbenchmark-based repository optimization	Performance PRs and manually reconstructed workloads	Final-only microbenchmark-based evaluation
FormulaCode (Sehgal et al., 2026)	Microbenchmark-based repository optimization	Continuously ingested performance PRs and ASV workloads	Final-only microbenchmark-based evaluation

B.2 DETAILED BENCHMARK COMPOSITION

We characterize the 100 PERFRACE tasks by project purpose, primary programming language, license, packaged source size, file count, and fixed workload.

Projects. The projects represented in PERFRACE span 16 primary programming languages: Rust (25 tasks), Python (18), TypeScript (15), C++ (11), C (6), C# (6), Julia (5), Go (4), Java (3), Scala (1), Lua (1), OpenSCAD (1), Solidity (1), MATLAB (1), JavaScript (1), and Zig (1). The packaged project sources range from 42 to 15,869 files and from 471.49 KiB to 689.08 MiB, with medians of 886 files and 14.56 MiB.

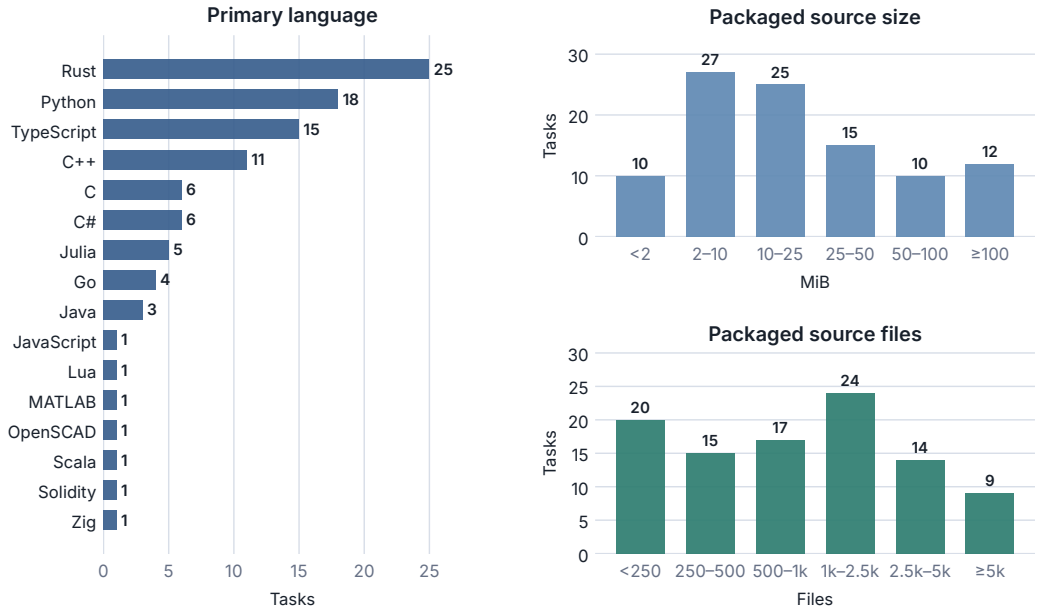


Figure 13: Distribution of primary programming languages, packaged source sizes, and packaged source file counts across PERFRACE projects.

Task catalog. Table 7 lists each task's project, purpose, license, and fixed workload.

Table 7: Complete catalog of the 100 PERFRACE tasks.

Task	Project and purpose	License	Fixed workload
`aiidateam-aiida-core--battery-archive-import`	`aiidateam/aiida-core` Scientific workflow and provenance platform	MIT	AiiDA import of a public battery-cycling provenance archive
`anyrender--openmoji`	`DioxusLabs/anyrender` Rust 2D drawing abstraction	Apache-2.0	AnyRender batch rasterization of the OpenMoji SVG corpus
`apalache--zcash-sync`	`apalache-mc/apalache` Symbolic model checker for TLA+ and Quint	Apache-2.0	Apalache's bounded safety check for the Zcash v2 block-download scheduler
`arrow-rs--variant-wikidata`	`apache/arrow-rs` Rust implementation of Apache Arrow	Apache-2.0	apache/arrow-rs while encoding Wikidata entities as a Variant column
`bevy-voxel-world--terrain-streaming`	`splashdust/bevy_voxel_world` Voxel-world plugin for Bevy	Apache-2.0	splashdust/bevy_voxel_world terrain generation and chunk meshing
`bm25s--trec-covid-index`	`xhluca/bm25s` BM25 search library	MIT	bm25s indexing and persistence for the TREC-COVID research collection
`boyter-scc--linux-v6-19-json-scan`	`boyter/scc` Source-code counting and complexity tool	MIT	SCC whole-repository JSON scan time on Linux v6.19
`calldiff--pi-review`	`tanishqkancharla/calldiff` Function call-stack diff tool	MIT	Calldiff JSON call-stack diffs for a real Pi coding-agent review
`cbcoutinho-nextcloud-mcp-server--ohr-pdf-ingest`	`cbcoutinho/nextcloud-mcp-server` MCP server for Nextcloud	AGPL-3.0	Nextcloud MCP Server's born-digital PDF ingest and page-aware chunking throughput

Continued on next page

Table 7 (continued)

Task	Project and purpose	License	Fixed workload
ccc--compile-speed	anthropics/claudes-c-compiler Dependency-free C compiler	CC0-1.0	CCC compile-to-object latency on the SQLite 3.46.0 amalgamation
chaste--ttp06-sheet	Chaste/Chaste Biomedical simulation framework	BSD-3-Clause	Chaste cardiac simulation for a TTP06 ventricular sheet propagation
cmtraceopen--hadoop-logs	adamgell/cmtraceopen Cross-platform Windows log viewer	MIT	CMTrace Open while loading a real Hadoop/YARN log session
cocoon--oci-import	cocoonstack/cocoon MicroVM engine	MIT	local import of a supported Cocoon Ubuntu VM OCI image
codediff--neovim-release	esmuellert/codediff.nvim Neovim code-review and diff plugin	MIT	CodeDiff C diff latency for a real Neovim release review
commitizen--check	commitizen-tools/commitizen Commit and changelog automation tool	MIT	Commitizen changelog generation for Angular's complete Git history
core--blob-upload	calimero-network/core Peer-to-peer application platform	Apache-2.0	Calimero content-addressed blob ingestion for a real OpenStreetMap file upload
cpp2rust--woff2-compress	Cpp2Rust/cpp2rust C++-to-safe-Rust translator	MIT	Cpp2Rust's safe Rust output for public-font WOFF2 compression
deepinv--kohler-kernels	deepinv/deepinv Deep-learning inverse-problems library	BSD-3-Clause	DeepInverse spatially varying blur-kernel estimation on a real Kohler camera-motion-blurred image
dftd4--lysozyme-gradient	dftd4/dftd4 Atomic-charge-dependent dispersion correction	LGPL-3.0	DFT-D4 energy, analytic gradient, and virial evaluation for a real lysozyme structure
dither-guy--photo-batch	manoelpiovesan/dither-guy Image dithering application	GPL-3.0	Dither Guy's full-resolution photo batch export
doc-detective--redpanda-transforms	doc-detective/doc-detective Documentation testing framework	AGPL-3.0	Doc Detective dry-run planning for Redpanda transform documentation
drasi-core--gtfs-delay-replay	drasi-project/drasi-core Continuous-query engine	Apache-2.0	Drasi Core's continuous Cypher evaluation for replayed GTFS-Realtime vehicle geofence updates
duckdb--taxi-ingest	Giorgi/DuckDB.NET .NET bindings for DuckDB	MIT	DuckDB.NET ingestion of NYC Yellow Taxi trip records
dusty--fpindex-ingest	lalinsky/dusty Zig HTTP client/server library	MIT	Dusty's HTTP batch-ingestion path for real AcoustID fpindex updates
embree--chess-render	RenderKit/embree Ray-tracing kernels	Apache-2.0	Embree triangle traversal for a public chess-scene visibility render
endee--esci	endee-io/endee Vector database	AGPL-3.0	Endee HTTP dense search for a replay of Amazon Shopping Queries product searches
enzyme--mnist	EnzymeAD/Enzyme.jl Julia automatic-differentiation library	MIT	Enzyme.jl for Flux's CPU LeNet MNIST training epoch
eole--gpt2-wikitext-generate	eole-nlp/eole Language-modeling toolkit	MIT	EOLE CPU GPT-2 batch text generation on WikiText-103 prompts
eucalypt--config-render-speed	curvelogic/eucalypt Configuration language and interpreter	MIT OR Apache-2.0	the eucalypt interpreter on a config-generation workload
eucalypt--eval-speed	curvelogic/eucalypt Configuration language and interpreter	MIT OR Apache-2.0	the eucalypt interpreter on an Advent of Code workload
facet--venta-guild	Tim-Maes/Facet .NET DTO and mapping source generator	MIT	Facet package and source-generator compilation time for venta's Guild service
flexmeasures--smard-price-import	FlexMeasures/flexmeasures Energy-management platform	Apache-2.0	FlexMeasures import of a full year of public SMARD day-ahead prices
fluidsynth-fluidsynth--bagatelles-midi-render	FluidSynth/fluidsynth SoundFont software synthesizer	LGPL-2.1	FluidSynth offline rendering of a complete CC0 MIDI performance to WAV
geoparquet-io--hilbert-sort	geoparquet/geoparquet-io GeoParquet tooling	Apache-2.0	GeoParquet Hilbert ordering of an Overture Buildings shard
gridflock--drawer-stl	yawkat/GridFlock Gridfinity baseplate generator	MIT / CC-BY-4.0	GridFlock's segmented Gridfinity drawer-baseplate STL export

Continued on next page

Table 7 (continued)

Task	Project and purpose	License	Fixed workload
ifcopenshell--gni-bimproject-obj-export	IfcOpenShell/IfcOpenShell IFC library and geometry engine	LGPL-3.0	IfcOpenShell IfcConvert OBJ export for a lighter GNI BIM Project architectural IFC delivery
intro-skipper--credits	intro-skipper/intro-skipper Jellyfin intro and credits detector	GPL-3.0	Intro Skipper's movie credits black-frame evidence extraction
jagua--nesting	JeroenGar/jagua-rs 2D collision engine for nesting	MPL-2.0	jagua-rs collision collection replayed from a downstream nesting solver
javaevolved--site-build	javaevolved/javaevolved.github.io Java developer documentation site	MIT	the full localized GitHub Pages build for java.evolved
jsonnet--kubernetes-mixin-speed	google/jsonnet Data templating language	Apache-2.0	the Jsonnet C++ interpreter on a Kubernetes monitoring configuration workload
juliaio-zarr-jl--oceananigans-timeseries-output	JuliaIO/Zarr.jl Julia implementation of Zarr array storage	MIT	Zarr.jl directory-store output for a real Oceananigans tracer time series
julialang-julia--trixi-jeans-instability	JuliaLang/julia Julia programming language	MIT	Julia runtime execution for a published Trixi Jeans gravitational-instability simulation
kitops--minilm-pack	kitops-ml/kitops OCI packaging tool for ML artifacts	Apache-2.0	KitOps local ModelKit packaging for a complete all-MiniLM-L6-v2 release snapshot
kiwifs--kubernetes-reindex	kiwifs/kiwifs Markdown filesystem for agents and teams	BUSL-1.1	KiwiFS SQLite FTS5 reindex for Kubernetes documentation
kms--batch-encrypt	Cosmian/kms Key-management system	BUSL-1.1	Eviden KMS KMIP BulkData AES-GCM encryption for concurrent application-record batches
kubb--esi	kubb-labs/kubb OpenAPI code-generation framework	MIT	Kubb regeneration of JitaSpace's EVE Online ESI client
latexsnipper--formula-batch	SakuraMathcraft/LaTeXSnipper Formula OCR and editing tool	GPL-3.0	resident CPU formula OCR for a public OpenStax textbook screenshot batch
libcachesim--twitter-lru	1a1a11a/libCacheSim Cache simulation library	Apache-2.0	Four-capacity LRU cache-size sweep over a production Twitter request trace
lsp-tree-sitter--termux-check	neomutt/lsp-tree-sitter Tree-sitter language-server library	GPL-3.0	Tree-sitter validation of a real Termux package-recipe corpus
mai-gen-videob50--score-images	Nick-bit233/mai-gen-videob50 Maimai score-video generation tool	MIT	batch maimai score-image rendering for a real 50-record archive
meshmonitor--coverage-grid	Yeraze/meshmonitor Mesh-node monitoring web tool	BSD-3-Clause	MeshMonitor's SQLite coverage-grid batch
metamorpho--reallocate	morpho-org/metamorpho Tokenized lending-vault protocol	GPL-2.0	Gas use for a historical MetaMorpho market reallocation
mini-qr--library-batch	lyqht/mini-qr QR-code generator and scanner	GPL-3.0	Mini QR's batch SVG export for a public library directory
mirage--kubernetes-triage	strukto-ai/mirage Virtual terminal for coding agents	Apache-2.0	Mirage recursive source search for Kubernetes dual-stack readiness-probe triage
modflow6--neversink	MODFLOW-ORG/modflow6 Groundwater-flow simulator	CC0-1.0	Complete steady-state simulation of the USGS Neversink–Rondout groundwater model
monorepo--crypto-build	commonwarexyz/monorepo Distributed-systems primitives	Apache-2.0	the release build latency of commonware-cryptography
mrst--norne-import	SINTEF-AppliedCompSci/MRST Reservoir-simulation toolbox	GPL-3.0	MRST Norne field-model import and geometry construction
msdialworkbench--lcms-alignment	systemsomicslab/MsdialWorkbench Mass-spectrometry analysis workbench	LGPL-3.0	MS-DIAL Console LC-MS peak processing and alignment for a real negative-ion DDA batch
mssql-extension--taxi-copy	hugr-lab/mssql-extension DuckDB extension for Microsoft SQL Server	MIT	BCP bulk import of NYC taxi records through the DuckDB MSSQL extension
n8n-mcp--tool-batch-speed	czlonkowski/n8n-mcp MCP server for n8n workflow construction	MIT	n8n-mcp on a fixed MCP tool-call batch

Continued on next page

Table 7 (continued)

Task	Project and purpose	License	Fixed workload
nerdbank-messagepack--earthquake-fanout	AArnott/Nerdbank.MessagePack .NET MessagePack library	MIT	SignalR MessagePack earthquake snapshot fan-out
ojp--taxi-export	Open-J-Proxy/ojp Database proxy and JDBC driver	Apache-2.0	OJP JDBC export of public NYC TLC taxi records
omnigraph--metabolic-trials-refresh	ModernRelay/omnigraph Lakehouse-native graph engine	MIT	OmniGraph's paginated metabolic clinical-trial refresh and lifecycle dashboard
onnxscript--bert-optimize	microsoft/onnxscript ONNX model authoring and optimization library	MIT	Graph optimization of a BERT-SQuAD FP32 and INT8 model bundle
oxidize-pdf--rag-chunking	bzsanti/oxidizePdf Rust PDF library for RAG	MIT	oxidize-pdf RAG chunking over a real CC-BY PDF corpus
pdal--dtm	PDAL/PDAL Point-cloud processing library	BSD-3-Clause	Ground classification and 1 m terrain-model generation from USGS 3DEP LiDAR
petsc--ocean-age-solve	JuliaParallel/PETSc.jl Julia wrappers for PETSc	MIT	PETSc.jl GMRES+ILU for a public OCCA ocean-age transport solve
physical-ai-toolchain--action-smoothness	microsoft/physical-ai-toolchain Physical-AI dataset analysis toolkit	MIT	multi-dataset LeRobot action-smoothness analysis for real ALOHA manipulation recordings
plonky3--ethereum-calldata-keccak-proof	Plonky3/Plonky3 Polynomial IOP toolkit	Apache-2.0	Plonky3 Ethereum calldata Keccak trace generation and STARK proving
polkadot-api--descriptor-generation	polkadot-api/polkadot-api Modular Polkadot API libraries	MIT	PAPI's multi-chain TypeScript descriptor generation
pytrendy--dow-index	RussellSB/pytrendy Trend-detection library	MIT	PyTrendy trend detection for a real daily Dow Jones Industrial Average price series
qrisp--modmul-compile	eclipse-qrisp/Qrisp Quantum-algorithm development framework	EPL-2.0	Qrisp compilation of controlled modular multiplication
react-native-code-push--mattermost-release	Soomgo-Mobile/react-native-code-push React Native OTA update client	MIT	CodePush Android OTA releases for Mattermost Mobile
reactant--fashion-mnist-train	EnzymeAD/Reactant.jl MLIR/XLA compiler for Julia	MIT	Reactant CPU execution of a real Lux Fashion-MNIST training epoch
rspress--react-native-paper	web-infra-dev/rspress Static-site generator	MIT	Rspress production builds of the React Native Paper documentation site
rust-clippy--cargo-package-lint	rust-lang/rust-clippy Rust linter	Apache-2.0	Clippy lint time for Cargo's main production package
s3s--kubernetes-sync	s3s-project/s3s S3 service adapter	Apache-2.0	s3s-fs recursive S3 synchronization of Kubernetes pkg source
senpi--image-batch	code-yeongyu/senpi Coding-agent runtime	MIT	Senpi image attachment preprocessing for a real screenshot batch
snarkos--cdn-replay	ProvableHQ/snarkOS Decentralized operating system for ZK applications	Apache-2.0	snarkOS CDN block bundle replay over 150 fixed Aleo Mainnet blocks
snarkvm--mainnet-execution-verify	ProvableHQ/snarkVM Zero-knowledge virtual machine	Apache-2.0	snarkVM verification of a batch of accepted Aleo mainnet executions
solvespace-solvespace--printer-assembly-mesh-export	solvespace/solvespace Parametric 2D/3D CAD	GPL-3.0	SolveSpace CLI mesh export for a complete Cartesian 3D-printer assembly
spectral--api-lint	stoplightio/spectral OpenAPI, JSON, and YAML linter	Apache-2.0	Spectral linting of a production OpenAPI batch
sqruff--glam-etl	quarylabs/sqruff SQL formatter and linter	Apache-2.0	Sqruff directory linting for Mozilla GLAM ETL SQL
strawberry-graphql-strawberry--graphique-tlc-weekly-columns	strawberry-graphql/strawberry Python GraphQL library	MIT	Strawberry GraphQL execution for Graphique's real TLC weekly Parquet column query
suitesparse--livejournal-pagerank	DrTimothyAldenDavis/SuiteSparse Sparse-matrix and graph computation suite	Apache-2.0 / BSD-2-Clause (used components)	Complete PageRank computation on the LiveJournal social graph using GraphBLAS and LAGraph
sum--text-attest	OtotaO/SUM Cross-runtime text-attestation library	Apache-2.0	SUM while batch-attesting a corpus of real Wikipedia prose
sundials--cantera-ignition	llnl/sundials Nonlinear and differential-equation solver suite	BSD-3-Clause	SUNDIALS CVODES for a downstream Cantera methane ignition parameter sweep

Continued on next page

Table 7 (continued)

Task	Project and purpose	License	Fixed workload
svf-tools-svf--nginx-andersen-wpa	SVF-tools/SVF Static value-flow analysis framework	AGPL-3.0-or-later	SVF Andersen whole-program pointer analysis on NGINX reverse-proxy bitcode
synalinks--kubernetes-tasks	SynaLinks/synalinks Neuro-symbolic language-model framework	Apache-2.0	Synalinks Markdown knowledge-base ingestion for Kubernetes task documentation
tfhe-rs--erc7984-transfer	zama-ai/tfhe-rs Fully homomorphic encryption library	BSD-3-Clause-Clear	TFHE-rs CPU execution for confidential ERC-7984 token transfers
tileserver-gl--country-atlas	maptiler/tileserver-gl Vector and raster map-tile server	BSD-2-Clause	TileServer GL static PNG rendering for a full Natural Earth country atlas
trixi-framework-trixi-jl--self-gravitating-sedov-amr	trixi-framework/Trixi.jl Adaptive numerical-simulation framework	MIT	a published self-gravitating Sedov blast-wave simulation with adaptive mesh refinement
typst--napkin	Myriad-Dreamin/typst.ts Typst runtime for JavaScript	Apache-2.0	Typst.ts rendering in the Napkin Community production static-site build
vmecpp--precise-qh	proximafusion/vmecpp Magnetohydrodynamic equilibrium solver	MIT	VMEC++ fixed-boundary precise-QH equilibrium solves
wiredtiger--pageviews	wiredtiger/wiredtiger Embedded storage engine	GPL-2.0 OR GPL-3.0	WiredTiger's durable import of a real Wikimedia pageview hour
xet-core--bert-download	huggingface/xet-core Content-addressed transfer and reconstruction engine	Apache-2.0	local Xet reconstruction of a production BERT model artifact
yamato-security-hayabusa--external-attack-evtx-timeline	Yamato-Security/hayabusa Windows event-log forensics tool	AGPL-3.0	Hayabusa bulk external attack-EVTX JSONL timeline generation
youtrackdb--osm-roads	JetBrains/youtrackdb Object-oriented graph database	Apache-2.0	embedded YouTrackDB import of a real OpenStreetMap road network
zenbu--release-source	zenbu-labs/zenbu.js JavaScript application framework	MIT	Zenbu.js staging of the Zenbu desktop application's release source
zio--zrk-typecheck	lalinsky/zio Zig asynchronous-I/O framework	MIT	ZIO's downstream zrk ReleaseFast type-check latency

B.3 Construction and Curation Statistics

This section describes the implementation of Discovery and Filter and reports the candidate counts and curation effort at each construction stage.

Discovery implementation. We query the ClickHouse `github_events` table for opened pull requests and branch pushes by a configured list of coding-agent accounts, then group matching events by repository. For each repository, we retain the latest matching pull-request or branch URL and use the GitHub GraphQL API to record its canonical identity, default-branch revision, stars, primary language, license, disk usage, and repository status.

The resulting records are deduplicated by GitHub repository, and projects are excluded when their repository, default branch, or recorded source revision cannot be retrieved. Each step writes its results incrementally, allowing interrupted collection and metadata retrieval to resume without repeating completed work.

From GitHub activity recorded between May and August 2026, we identified 118,517 repositories, retrieved metadata for 104,428, and retained 103,475 project candidates after these checks.

Table 8: GitHub activity signals used for Discovery.

Signal	Repositories
Branch push by a coding-agent account	112,127
Pull request opened by a coding-agent account	6,390

Filter implementation. We first apply deterministic checks to repository metadata and the recorded Git tree. These checks require an accessible source revision, a license permitting modification and redistribution, 10–4,999 GitHub stars, and a tracked Git tree of at most 500 MiB and 25,000 entries.

Projects that pass these checks are collected at the recorded revision with their README, setup documentation, package manifests, license files, and representative source paths. We then use GPT-5.6-Luna to assess the project's setup path, evaluable process, runtime fit, and optimization potential from this source snapshot using a structured response schema. The prompt is provided in Appendix D.1.

We require each structured response to address the following criteria.

- ☐ **Setup path.** Repository files support concrete commands for installing dependencies and building or running the project.
- ☐ **Evaluable process.** The project supports a finite process with a fixed input, observable output, performance metric, and correctness check.
- ☐ **Runtime fit.** The process can run in a Linux x86-64 container on commodity CPUs without mandatory special hardware.
- ☐ **Optimization potential.** The project supports a substantial local computation whose cost is expected to increase with input size or repeated execution.

Using these assessments, we selected 239 high-potential project candidates for Pair Verification. Table 9 reports the first check that excludes each project from the final Pair Verification selection.

Table 9: First failed check during Filter and the final selection result.

Result	Projects
Fewer than 10 GitHub stars	93,992
Accessibility	1,134
Setup path	212
Runtime fit	359
Optimization potential	4,462
Evaluable process	3,066
Assessment error	11
Selected for Pair Verification	239

Curation effort. To quantify the effort required to turn selected projects into accepted tasks, Figure 14 reports the agent cost, curation time, and number of agent–human interactions for the 100 accepted tasks. The median task required 4.5 hours, $8.83 in agent cost, and eight agent–human interactions, compared with averages of 9.1 hours, $11.35, and 12 interactions.

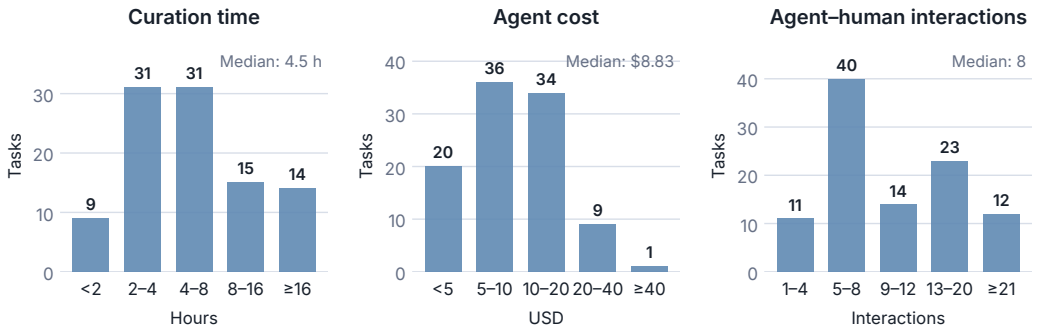


Figure 14: Distribution of curation time, agent cost, and agent–human interactions across the 100 accepted tasks.

B.4 Pair Verification

PerfRace uses Pair Verification to ensure that each task represents meaningful project use, produces reproducible measurements, and rejects shortcut solutions; this section describes the agent–human protocol and analyzes the resulting curation trajectories.

B.4.1 Protocol

Pair Verification combines agent-led task construction with human review to turn each project candidate into an executable task and assess its realism, reliability, and integrity. The construction skill is provided in Appendix D.2. The agent develops the task in stages and pauses for a reviewer decision after completing the evidence required for each quality criterion.

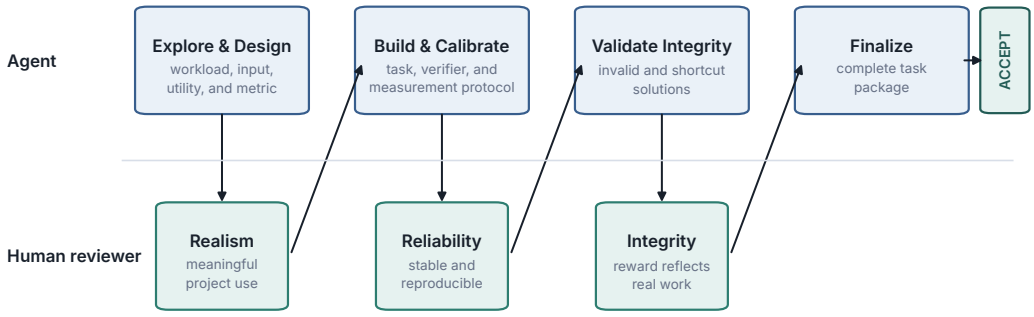


Figure 15: Pair Verification alternates between agent-led task construction and human review at the Realism, Reliability, and Integrity gates.

Explore and Design. The agent first studies the project's documentation, examples, code paths, and public usage evidence to understand its purpose, users, and performance-relevant operations. The agent searches public release pages for source code, models, and assets, scientific or government data portals, downstream repositories, production archives, and real user exports to identify candidate workload inputs. For each input, the agent records its download location, version, and applicable terms, and accepts it only when an explicit license, public-domain designation, or written authorization permits redistribution with the benchmark.

It uses these materials to identify candidate operations, but grounds the workload and scale in a real user or downstream application rather than repository-native tests, benchmarks, examples, or fixtures. The agent compares plausible workload candidates, then traces the selected operation from its user-facing entry point through the relevant project code and demonstrates a small input-to-output execution on the unmodified revision. During this run, it records dependency installation, project build, input preparation, and operation time to check that setup does not dominate the measurement and to estimate whether complete verification can finish within the 1,200-second budget. The agent repeats the operation on identical inputs to detect run-to-run flakiness, including sporadic failures or inconsistent correctness-relevant outputs, and to identify any dependence of runtime on random state, network access, or external services. A candidate is revised or rejected if its stochastic execution remains too variable for reproducible performance measurement after fixing available random seeds and aggregating repeated runs.

It then specifies the exact operation, workload and scale available to the agent, held-out correctness cases, observable correctness utility, performance metric, and measurement boundary. Before requesting Realism review, the agent summarizes the project's purpose and users, the selected execution path, the basis for the input and scale, and the alternatives it considered and rejected. The reviewer approves Realism only when the operation, input, and scale represent meaningful project use, the metric measures the stated utility, and the workload inputs can be redistributed as part of the benchmark.

Build and Calibrate. After Realism approval, the agent derives the installation, build, and run commands from upstream documentation, CI configuration, and package manifests, and verifies them on the pinned source revision. The agent implements the verifier, the task's test program that measures the performance metric on a separate copy of the workload and checks additional held-out cases for correctness. It then constructs the Harbor task files with `instruction.md`, `task.toml`, an `environment/` directory containing the editable project and workload, and a separate `tests/` directory containing the verifier code, workload copy, held-out cases, and expected outputs. The agent pins the project revision, base image, language toolchain, dependencies, workload assets, and expected outputs so that the image can be rebuilt and the task rerun without downloading project source or workload inputs. The task instruction tells the agent how to build and run the project, what it may edit, which behavior it must preserve, and which performance metric to improve.

The agent then builds a fresh image and runs the verifier on the unmodified project before calibrating performance measurements. The agent selects the smallest resource profile that completes the workload without CPU, memory, or storage pressure and sets the verifier timeout within 1,200 seconds. The agent uses pilot and repeated runs to choose the workload scale, warmups, repetitions, and aggregation, requiring the final measurements to have a coefficient of variation below 10% while keeping build and setup outside the timed operation. The agent checks that the timing boundary covers the complete operation requested in the task while excluding one-time environment setup and verifier preparation that are unrelated to the optimization objective.

For the n timed measurements retained after warmup, $x_1, \ldots, x_n$, we compute

$$\mathrm{CV} = \frac{s}{\bar{x}}, \qquad \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}.$$

After fixing the workload, correctness checks, resources, and measurement protocol, the agent builds the final image and evaluates the unmodified baseline and one optimization candidate under the same conditions. The reviewer examines the fresh-build correctness results, repeated measurements and CVs, verification time, resource allocation, and candidate trajectory. The reviewer approves Reliability only when fresh builds and repeated runs return the same correctness verdict and stable performance measurements with the chosen warmups, repetitions, aggregation, and measurement boundary.

Validate Integrity. To identify reward-hacking paths that could receive a valid score without completing the task, the agent tests deliberately incorrect, partial, hard-coded, and work-skipping solutions and checks that workload inputs, expected outputs, and verifier-owned data cannot be

replaced by agent-controlled files. The agent inspects the evaluation image for leaked expected outputs, verifier code, or construction notes that could reveal the hidden checks. For example, in `rust-clippy-cargo-package-lint`, removing Cargo dependency tracking made later runs reuse a cached lint result, which the verifier detects by touching `clippy-driver`, running the workload twice against the same warm target, and requiring each Cargo JSON trace to report a non-fresh compiler artifact for `cargo`.

The reviewer approves Integrity only when every requirement in the task instruction is checked, every verifier check follows from the declared task, and none of the tested shortcut solutions can receive a valid score.

Finalize. After all three reviews, the reviewer chooses ACCEPT, REVISE, or REJECT; accepted tasks pass final compatibility checks before publication as a task package comprising an immutable Docker image and task artifact.

B.4.2 Quality Control

We analyze the recorded agent–human Pair Verification trajectories for evidence of the Realism, Reliability, and Integrity of the resulting tasks. Of the 239 candidates, 132 were rejected during Explore and Design, six during Build and Calibrate, none during Validate Integrity, and one during Finalize, leaving 100 accepted tasks.

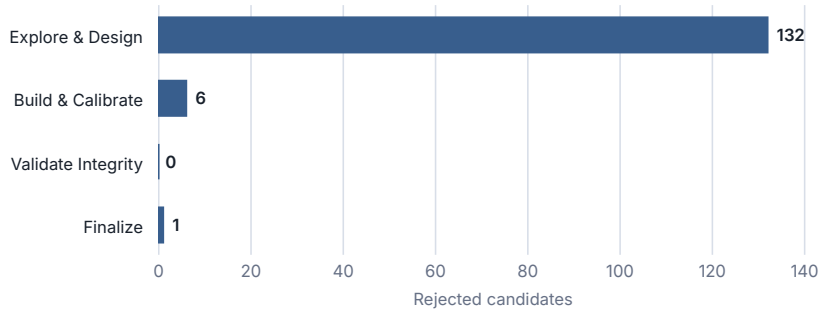


Figure 16: Candidates rejected at each Pair Verification stage, with candidates lacking a recorded transition into Build & Calibrate assigned to Explore & Design.

Realism. Realism checks cover the workload operation and scale, the source and redistribution terms of its inputs, and the alignment between the declared utility and performance metric. Table 10 groups the workload inputs into released source code, models, or assets (26 tasks), public scientific or government datasets (21), real downstream applications (21), production traces or archives (16), real user exports (6), and other independently grounded inputs (10).

Table 10: Sources of workload inputs across the 100 accepted tasks.

Input source	Examples	Tasks
Released source code, model, or asset	Linux source tree, OpenMoji assets, released ML models	26
Public scientific or government dataset	TREC-COVID, NYC taxi records, LC-MS data	21
Real downstream application	React Native Paper, Headlamp, Mattermost Mobile	21
Production trace or archive	Hadoop logs, GTFS delay feeds, GitHub events	16
Real user export	MIDI collection, BIM project, printer assembly CAD	6
Other independently grounded input	Wikidata variants, Dow index, OSM roads	10

Reliability. All 99 wall-clock tasks have a final unmodified-project CV below 10%; the remaining task, `metamorpho-reallocate`, measures transaction gas, the deterministic execution cost assigned by the Ethereum Virtual Machine to a fixed smart-contract transaction, and returns exactly 874,508 gas in three fresh containers. Table 11(a) shows that 84 of the 99 wall-clock tasks have a final CV below 6%.

Integrity. We validate Integrity across six attack classes: empty or no-op solutions, skipped work, hard-coded outputs, precomputed or cached outputs, modification of protected inputs or tests, and access to verifier-only data. All 100 accepted tasks completed an explicit red-team pass before Integrity approval. Table 11(b) shows that 40 tasks were tested against at least two red-team cases, including two tasks with seven or eight cases. For example, SCC is a source-code statistics tool whose task scans the Linux v6.19 tree and emits JSON statistics; during red-team testing, the agent replaced the scanner with a program that returned the precomputed Linux result instead of scanning the source tree, which passed the workload available to the agent but failed when the verifier asked it to scan a separate pinned checkout of SCC's own source code.

(a) Final measurement CVs

CV range	Tasks
$0\% \leq \text{CV} < 2\%$	20
$2\% \leq \text{CV} < 6\%$	64
$6\% \leq \text{CV} < 7\%$	8
$7\% \leq \text{CV} < 8\%$	3
$8\% \leq \text{CV} < 9\%$	4
$9\% \leq \text{CV} < 10\%$	0

(b) Red-team tests

Tests per task	Tasks
1	60
2	20
3	12
4	6
7	1
8	1

Table 11: Reliability and integrity evidence: final CVs for the 99 wall-clock tasks and red-team tests for all 100 tasks, with one test assigned conservatively when an exact count cannot be recovered.

B.5 Evaluation Protocol

For each submission, the Judge snapshots the agent's current container and starts a temporary shadow container from that snapshot. The Judge creates the snapshot with `docker compose commit`, launches the shadow container under a separate Compose project, and removes its container, local image, and volumes after verification. The agent container remains paused during verification and resumes from the same workspace state after the shadow container is removed.

Inside the shadow container, the verifier runs through `tests/test.sh` and emits the correctness verdict and performance metric as JSON. The verifier code, workload copy, held-out cases, and expected outputs are absent from the agent container and available only inside the shadow container. Each task declares its warmups, repetition count, aggregation rule, and required correctness pass rate in `task.toml`, and its verifier returns a performance metric only when that pass rate is met.

The task clock continues while the shadow is created, executed, and removed, so every submission trades part of budget B for additional correctness and performance feedback. After verification, the Judge returns the correctness verdict and performance metric to the agent as feedback for subsequent optimization. For a correct submission, the Judge computes the speedup score by dividing the unmodified baseline metric by the submission metric. Submissions that fail correctness, exceed the verifier timeout, or do not produce a valid performance metric cannot update the best-so-far trajectory $S(t)$.

Five minutes before the deadline, the agent receives a final delivery prompt asking it to leave the project in a coherent state before the Judge performs one final shadow verification of the current workspace. Figure 17 characterizes the time required for one complete Judge evaluation, showing that baseline Judge evaluation remains within the 20-minute limit for all 100 final tasks.

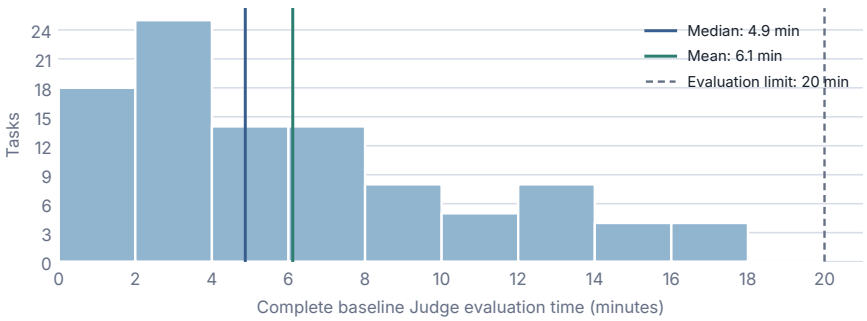


Figure 17: Distribution of complete baseline Judge evaluation time across the 100 final task packages, with a mean of 6.1 minutes, a median of 4.9 minutes, and a 20-minute evaluation limit.

Longer optimization budgets. To measure the benefit of additional optimization time, we run Terminus-KIRA with GPT-5.6-Luna at high reasoning effort three times on each of the 88 evaluated tasks, maintaining one persistent agent session across Judge submissions during every four-hour run.

The aggregate best-so-far correct speedup rises from $1.314\times$ after one hour to $1.377\times$, $1.396\times$, and $1.435\times$ after two, three, and four hours, respectively. Across the resulting 264 trajectories, Figure 18 shows that extending Terminus-KIRA's budget from two to four hours improves the overall score by 4.2%, and the fourth hour alone adds 2.8%.

The four-hour score is at least 5% higher on 24 tasks than at two hours and on 12 tasks than at three hours, showing that agents continue to find meaningful improvements late in the task. Among the 228 runs that improve over baseline, the best result is first reached at a median of 88.1% of the four-hour budget, and 51 of the 264 runs produce a new improvement of at least 5% during the final hour.

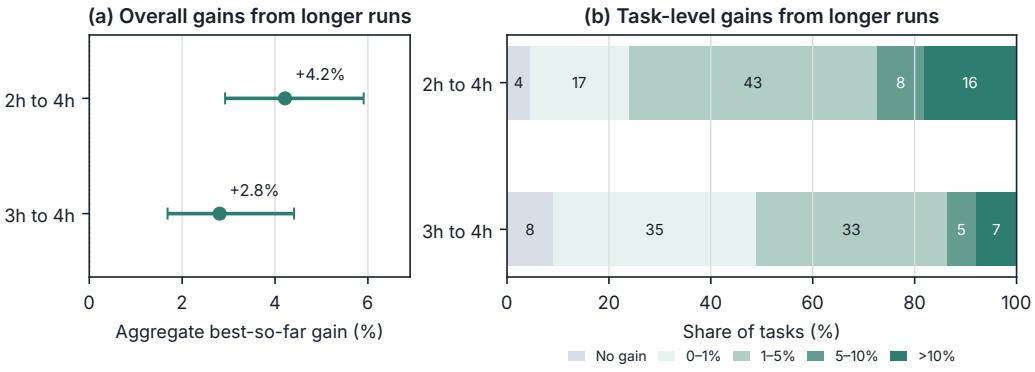


Figure 18: Performance improvements from additional optimization time, based on three four-hour runs for each of 88 PERFRACE tasks. Left: increase in the overall score at four hours relative to two and three hours; horizontal lines show 95% bootstrap intervals. Right: number of tasks in each improvement range.

C BoxBox: Method Details

Figure 19 summarizes the task and review loops. The following explains how harness revisions affect subsequent interactions and how review, hot reloading, and recovery preserve task continuity.

BoxBox

State: harness H, runtime R, task environment e, task trajectory τ.
Review state: environment e_r and separate trajectory τ_r.
Budget: shared task budget B.

while budget B remains **do**
  $a \sim p_{\mathcal{M},H}(\cdot \mid \tau)$
  if $a =$ `boxbox()` **then**
    Yield control to the kernel; suspend task execution and preserve context
    Prepare (e_r, τ_r) for independent review under H, with access to τ
    while true do
      $a_r \sim p_{\mathcal{M},H}(\cdot \mid \tau_r)$
      if $a_r =$ `commit`(H') **then break**
      $(e_r, o_r) \leftarrow \mathrm{Execute}_H(e_r, a_r)$
      $\tau_r \leftarrow \tau_r \oplus (a_r, o_r)$
    end while
    if $H' \neq H$ **then**
      The kernel validates H' and attempts activation of its runtime R'
      if successful **then** select $(H, R) \leftarrow (H', R')$
      else retain or restore the previous harness and runtime
    end if
    Restore continuation context and return control to the selected runtime
  else
    $(e, o) \leftarrow \mathrm{Execute}_H(e, a)$
    $\tau \leftarrow \tau \oplus (a, o)$
  end if
end while

Figure 19: **BoxBox pseudocode.** Task execution and independent harness review form nested interaction loops, with kernel-managed hot reloading connecting them.

C.1 Effects on Subsequent Interactions

Equation 1 describes two ways a harness revision can affect subsequent execution while the language model $\mathcal{M}$ remains fixed. Changes to how the harness constructs model inputs can alter $p_{\mathcal{M},H}$, influencing the agent's next action. Changes to $\mathrm{Execute}_H$ can alter how that action is carried out and how its result is returned as an observation. A revision can affect either or both, so its influence extends to the trajectory generated after activation.

The current harness also governs the review loop. Once a revision is activated, subsequent reviews use the updated implementation and can examine the experience accumulated under it. Harness changes can therefore affect both task execution and the process used to produce later revisions. Appendix A.6 illustrates concrete changes to guidance, measurement, command polling, and context handling.

C.2 Review

The task-solving agent requests review through `boxbox()` without having to propose a patch. The initial harness requires intervening task activity and a 30-minute cooldown after each completed review; once eligible, the agent decides whether and when to request another review. The kernel preserves the task conversation and launches a separate agent review session under the current harness. The reviewer can inspect task trajectories and Git history, but edits only the harness, treating

project code as read-only. It may retain the current harness, add guidance, or modify executable behavior, committing any changes before ending review. The default review skill is provided in Appendix D.3.

C.3 Execution Context

Execution context includes both runtime-owned state and the existing task environment. The former includes the task conversation, review history, and records of which trajectory entries have already been reviewed. These are exported as a state snapshot. The terminal session and project workspace remain in place: shell state, ongoing processes, and project files are preserved through the existing environment connection.

When the agent calls `boxbox()`, the runtime yields control to the kernel, suspending the task interaction loop while review proceeds separately. This pauses the agent's task decisions, not the entire container or its processes. Before importing state into a candidate runtime, the kernel makes a separate copy, preserving the original snapshot for recovery if the candidate fails. This separates changing the agent implementation from restarting its task environment.

C.4 Hot Reloading

Harness revisions are tracked as Git commits, while the kernel remains separate from the editable harness. Before switching, the kernel checks the committed candidate, loads its runtime, and verifies that it supports task execution and context transfer. The current runtime keeps its terminal connection until the candidate successfully imports the copied state. The kernel then transfers the same terminal session to the candidate and returns control to it, resuming task interaction from the preserved conversation in the same workspace. If activation fails, the kernel retains or restores the previous runtime and context. For agent-requested reviews, the reviewer can leave concise guidance on using new capabilities and continuing the task. This guidance is preserved across the update, and the resumed agent is prompted to read it before continuing. Task execution and review share the same budget and cost accounting. Table 12 summarizes how execution continues under each outcome.

Table 12: Task continuation across review and runtime-update outcomes.

Outcome	Continuation
No harness change	Resume the current runtime with the preserved task context.
Successful update	Continue under the revised harness with transferred context, the existing terminal session, and the same project workspace.
Update fails	Retain or restore the previous runtime and its preserved context.
Runtime fault	Enter recovery and review using the retained execution records; restart the terminal while preserving project files.

C.5 Recovery

The kernel retains a recovery point across harness updates. A newly activated runtime remains provisional until it reaches the next agent-requested review without a runtime fault. If it fails before that boundary, the kernel restores the previous confirmed harness and state and initiates a review using the fault evidence. Faults in a confirmed runtime also trigger review. Runtime-fault recovery starts a fresh terminal, so shell-local state is lost, but project files, harness history, and execution records remain available. The retained records let subsequent reviews examine failed revisions and help the task-solving agent determine how to continue interrupted work.

D PROMPTS AND SKILLS

This appendix presents the Filter prompt and Pair Verification skill used in benchmark construction, followed by the default BOXBOX self-review skill.

D.1 FILTER PROMPT

The Filter assesses repository snapshots before Pair Verification.

Filter Prompt

```
You are the automated Filter for PerfRace project candidates. Decide
whether the supplied repository snapshot provides enough source-grounded
evidence to proceed to Pair Verification.

Evaluate four gates:

1. Accessible. The repository and revision are available, with an
explicit license permitting modification and redistribution. A fork must
have an independently maintained product identity.

2. Upstream reproducibility. Documentation, scripts, or manifests
support concrete setup/build and minimal smoke commands. Cite the
supporting path or manifest field for each command. Dependencies and
configuration must come from identifiable public sources.

3. Evaluation potential. Identify a finite project operation with fixed
inputs, clear start and end, observable output, a plausible efficiency
metric, and a correctness check. Establish a replayable real user or
downstream workload, or a self-contained computation with grounded input
semantics. Repository tests, benchmarks, demos, fixtures, and generated
inputs do not establish a portable workload. Flag reliance on
unavailable deployment state, private traces, physical devices, or an
unresolved workload source.

4. Resource feasibility. The operation must run in a Linux x86-64
container on commodity CPUs within ordinary resource limits. Reject
mandatory special hardware, unavailable private services, and thin
remote pass-throughs. Local services or fixed stubs are acceptable only
when meaningful project-controlled computation remains.

Evidence and optimization potential:
- Use only the supplied snapshot; cite metadata or repository paths for
every gate. Missing evidence fails the gate.
- Describe a preliminary process only when evaluation potential passes.
- Describe the local computation and its strongest optimization
opportunity. Rate potential as low for tiny or externally dominated
work, medium for a clear local hot path, and high for scale-sensitive
computation with multiple structural opportunities.
- Return concise evidence in the requested structured response.
```

D.2 PAIR VERIFICATION SKILL

The skill guides agent-led construction through human reviews of Realism, Reliability, and Integrity, followed by task publication.

Pair Verification Skill

```
# PerfRace Pair Verification
```

Pair Verification Skill *(continued)*

```
Turn one repository into a machine-evaluable task. The agent
investigates and executes; the human reviews Realism, Reliability, and
Integrity, then decides ACCEPT, REVISE, or REJECT.

## Working agreement

Maintain a report linking claims to project sources, runnable probes,
and raw results. At each review gate, present the evidence, uncertainty,
and recommendation, allow discussion, and pause for the human decision.
Reopen the affected reviews when later evidence invalidates an approved
claim.

## Stage 1: Explore & Design

Understand the project's users and performance-sensitive workflows.
Compare candidate workloads and select an independently grounded real
user or downstream operation with reproducible inputs and scale.
Upstream examples and benchmarks may guide discovery but must not become
the scored workload.

Run the unmodified workload end to end and identify the project
execution path. Estimate build cost, image size, verification time, and
measurement variance before constructing the task. Propose the
operation, input provenance and scale, visible workload, independent
verifier copy, additional correctness cases, correctness utility,
measurement boundary, and metric.

Human review: Realism. Explain why the operation, inputs, and scale
represent meaningful project use, which alternatives were considered,
and what remains uncertain. Obtain approval before building the
canonical task.

## Stage 2: Build & Calibrate

Package the pinned project, visible workload, instructions, container,
and verifier. Keep only the project writable. Use an independent
byte-identical scored input and additional held-out correctness cases.
Correctness checks enforce the workload's observable utility, allowing
implementation changes that preserve it.

Keep candidate compilation and setup outside the timed user operation.
Calibrate workload scale, warmups, repetitions, resources, and timeout;
then freeze the measurement protocol. Complete verification must fit
within 1,200 seconds, and the uncompressed image must not exceed 3 GiB.
Target at least three seconds per timed execution; shorter operations
require a clear signal and repeated CV at most 10%.

Check fresh-image correctness and flakiness. Diagnose variance before
increasing workload size or resources. Enforce the same runtime policy
for baseline and candidate. Run a baseline and one bounded
candidate-agent attempt, retaining the trajectory and raw measurements.
One correct, repeatable gain is sufficient; an incorrect or no-gain
candidate does not reject the task if its trajectory demonstrates
understanding and a reasonable optimization attempt.

Human review: Reliability. Present fresh-build correctness, raw
measurements, dispersion, verification time, resources, and the
candidate trajectory. Ask whether equivalent runs support the same
verdict and measurement.

## Stage 3: Validate Integrity
```

Pair Verification Skill *(continued)*

```
Audit agreement between instructions and verifier checks, including
writable paths, workload copies, expected outputs, caches, and reward
inputs. Run explicit red-team tests for leaked-data access, verifier
modification, incomplete work, hard-coded answers, workload-specific
precomputation, frozen runtime-policy overrides, and reward-path
bypasses.

Count an attack only when a materially distinct invalid implementation
reaches the final verifier. After hardening, replay the attack and
record its verdict. Reopen Reliability if changes affect the workload,
measurement, resources, or verifier execution.

Human review: Integrity. Present the boundary audit and red-team
results. Ask whether an invalid or shortcut implementation can receive a
valid score. Verifier ownership requires an independent immutable
workload copy, not secrecy of the visible workload.

## Stage 4: Finalize

Summarize review decisions and limitations, remove construction
artifacts from the release package, and run final checks. Confirm that
the package contains all required inputs and no credentials, private
evidence, or hidden answers.

Ask the human to ACCEPT AND PUBLISH, REVISE, or REJECT. Publish the
pinned image and verified task package only after acceptance; preserve
evidence for rejected candidates.
```

D.3 Self-Review Skill

The following listing presents the default review skill with performance-optimization-specific guidance omitted. "Target Agent" denotes the task-solving agent.

Self-Review Skill

```
# Self-Review

Act as the macro-level controller of the Target Agent's task-solving
process. Review the trajectory to diagnose the task-solving process,
then retain the current harness or select a focused-feedback or
reusable-ability adaptation that improves subsequent Target progress
within the remaining shared budget.

## Review the task-solving process

Treat each review as one iteration in a continuing self-review loop. At
the start of every review, examine the accumulated trajectory
comprehensively before selecting an intervention.

Use that broad diagnosis to choose a focused intervention that is
supported by the evidence and likely to materially improve subsequent
Target progress. Include only the feedback or reusable capabilities
justified by the diagnosed problems in task execution and the evidence
needed to evaluate their effect.

Never be satisfied with the current optimization state while task budget
remains. Treat accepted improvements as intermediate evidence and
compare the remaining opportunities by their expected contribution.
```

Self-Review Skill *(continued)*

```
### Diagnose the task-solving process

Treat the Target Agent's framing, causal model, plans, explanations, and
claims as hypotheses, not premises. Use recorded actions, observations,
and measurements to identify all material unsupported links, distinguish
evidence from inference, and identify plausible explanations not ruled
out. Use the complete diagnosis to keep the intervention focused on the
related gaps that most materially constrain progress. Treat claims that
an experiment completed, improved performance, or excluded a direction
as unverified until supported by the trajectory. Explain how each
unsupported link affects task-execution cost, experiment selection, or
progress.

Critique the Target Agent's choice of actions, not just candidate
outcomes. Match each conclusion to the scope the experiments actually
tested. Do not generalize beyond the tested workload, configuration,
mechanism variant, or measurement, and do not ignore what the evidence
establishes within that scope. Distinguish direct findings from broader
hypotheses, and identify unsupported assumptions or omitted dimensions
that constrained the search.

After selecting the intervention focus, review two aspects of
macro-level control as ways to advance it:

1. **Task-execution cost.** Examine repeated Target Agent work, which
dependencies and evidence requirements have stabilized, and where
intermediate model judgment remains valuable. Look for opportunities to
reduce, fuse, automate, batch, or reuse work while preserving
decision-relevant evidence.
2. **Budget allocation.** Evaluate exploration breadth and exploitation
depth by comparing the causal support, remaining leverage, uncertainty,
and expected contribution of validated mechanisms and untested
hypotheses.

## Decide the intervention

After completing the diagnosis, choose among retention, a
focused-feedback adaptation, and a reusable-ability adaptation according
to their expected contribution to target progress over the remaining
budget. Retention means no new Target-facing guidance or other harness
change.

Use focused feedback for a task-specific concern, unsupported judgment,
omission, or unresolved evidence. Identify the concrete observation, the
gap, and why it matters. **Be careful about your own claims:** an
evidence gap does not establish an alternative explanation or direction;
do not turn preferences into concerns or next-step menus. State only the
concern and its evidentiary basis; do not require or recommend what the
Target Agent should do next.

Activate a reusable ability when it directly supports the diagnosis and
the trajectory supports repeated use. Also consider a harness adaptation
when the Target Agent repeatedly makes the same consequential error and
the evidence supports a general way to prevent it; otherwise use focused
feedback. For a stable repeated action sequence, fuse or automate the
mechanical work while preserving model judgment, decision-relevant
evidence, meaningful failure boundaries, and a fallback to the unfused
workflow. Validate the ability through its actual Target-facing path and
use feedback to explain how to use it and its constraints.
```

Self-Review Skill *(continued)*

```
Condition each intervention on the current stage of task execution,
inferred from the trajectory rather than elapsed time:

- **Early:** examine plausible competing hypotheses.
- **Developing:** reduce demonstrated repeated work and balance
exploration with supported exploitation.
- **Saturated:** identify which tested hypothesis families have low
remaining value, and flag any claim that the broader search is exhausted
unless the trajectory supports that scope.

Use the observed downstream effects of prior adaptations when deciding
what the current stage needs.
```