GOODHART'S LAW IN PERFORMANCE BENCHMARKS: MEASURING AND MITIGATING SOFTWARE OVEROPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents have shown growing promise for optimizing the performance of real-world software, motivating recent benchmarks that assess this capability through executable workloads. However, these benchmarks typically provide only one or a few visible workloads per task due to the substantial expert effort required to construct high-quality workloads. These workloads then act as both optimization feedback and the final evaluation. Agents can thus obtain large reported speedups by exploiting properties of the observed inputs rather than resolving the underlying performance bottleneck, producing patches that fail to generalize to other valid workloads. To systematically measure this workload overoptimization, we introduce WORKLOADPRO, an evaluation framework that expands existing benchmarks with diverse, held-out workloads. Measuring against these workloads shows that coding agents tend to specialize to the benchmark and produce inflated improvement numbers. To mitigate this problem, we introduce APO (**A**gentic **P**erformance **O**ptimization), a profile-guided agentic optimization framework that generates guardrail workloads at test time and explores diverse optimization targets against these workloads. Extensive evaluation using WORKLOADPRO shows that APO outperforms state-of-the-art open-source and proprietary agents and produces fewer overoptimized patches. We also applied APO to six open-source robotics systems and found 13 performance issues. Seven of them have been confirmed by the maintainers, and three patches have been merged. These findings show that workload diversity is important both for reliably evaluating coding agents and for guiding them toward performance optimizations that generalize.

1 INTRODUCTION

Coding agents have shown growing promise for repository-level performance optimization (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025a). Recent benchmarks assess this capability through executable workloads that measure the performance of agent-generated patches (Ma et al., 2026; Shetty et al., 2026; He et al., 2026). These workloads then serve as both optimization feedback and final evaluation. For example, a typical agent workflow repeatedly executes these workloads to obtain performance feedback that guides repository editing to minimize the execution time of the workloads, and the benchmark also relies on running these workloads to score the final patch (Ma et al., 2026).

This dual role becomes problematic when the visible workloads cover only a small or homogeneous set of inputs and executions. Related studies have also observed similar failures when agents repeatedly optimize against the limited tests in the coding benchmarks (Yu et al., 2025; Li et al., 2026a; Wang et al., 2025b; Ahmed et al., 2026). Unfortunately, performance tests (i.e., workloads) are more challenging to curate than the correctness tests used in typical coding benchmarks, as they measure properties beyond functional properties and often lack precise specifications (Yang et al., 2025). For example, each task in SWE-fficiency (Ma et al., 2026) contains only one workload, manually curated by human experts who ensured that the ground-truth performance patch meaningfully optimizes it. As a result, an agent can obtain a large measured speedup by exploiting incidental properties of this workload without resolving the underlying performance bottleneck. As a concrete example, Figure 1 shows that Claude Code games one task in SWE-fficiency by simply caching the execution to optimize the workload that stress-tests a color-conversion function in `matplotlib` by repeatedly

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.

invoking it with the same tuple object. Figure 1(b) further shows *overoptimization* when the agent is allowed to iteratively optimize against this workload, so the performance of its generated patch degrades on held-out workloads, e.g., those that do not use the same object, rendering the caching ineffective.

Challenge. An immediate mitigation is to augment the benchmarks by generating more diverse workloads, so that overoptimized patches are less likely to survive across all workloads. However, patches are not the only side that can overoptimize. While the generated patches can overfit to a single workload, simply generating more workloads based on a ground-truth patch can also over-specialize them on behaviors that the ground-truth patch happens to change. Simply optimizing both sides against each other can thus create a closed feedback loop in which the patch and workloads validate each other without representing the broader optimization opportunity. The key challenge is thus to improve both patches and workloads while preventing either side from defining the other's success. Specifically, we identify three properties that a performance benchmark should satisfy to reliably measure agentic code optimization. (1) Its workloads must be held out from the evaluated agents so that their measurements cannot guide patch optimization. (2) They should vary input values, shapes, types, workload structures, and execution paths to test whether a speedup generalizes beyond the observed cases. (3) Every workload must remain relevant to the intended optimization. The last two properties can create tension, where conditioning workload generation on a particular ground-truth patch can preserve relevance but produce patch-specific workloads, but unconstrained generation can yield diverse but unrelated workloads to the intended optimization.

Benchmarks. To this end, we introduce WORKLOADPRO, a diverse, held-out evaluation framework that disentangles workload generation from patch-based relevance verification. For each task, WORKLOADPRO iteratively constructs two complementary families of workloads. *Robustness* workloads vary the values, shapes, and types of inputs while preserving the original workload pattern. *Diverse* workloads vary the workload structure and must reach program paths not covered by previously accepted workloads. Only after a candidate has been generated is the ground-truth patch used to verify that the workload exercises the intended optimization, serving as a guardrail for workload generation rather than being included in the generation.

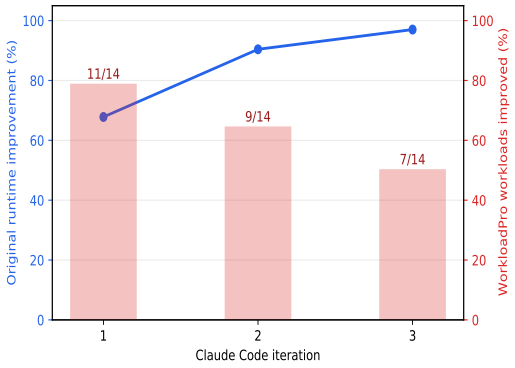
Our approach. Beyond using WORKLOADPRO to expose workload overoptimization, we introduce APO, a profile-guided agentic optimization framework that mitigates the overoptimization without accessing the held-out workloads in WORKLOADPRO. Our key approach is also based on workload generation in building WORKLOADPRO, but without accessing the ground-truth patch or an existing agent-generated patch. Specifically, before producing any candidate patch, APO generates diverse workloads from the original workload, relevant source code, and profiling information. It then fixes these workloads and profiles both the original and generated workloads to identify multiple distinct optimization targets, so the generated workloads both guard against overoptimization and broaden the evidence for locating the underlying performance bottleneck. This broader evidence is important as existing agents often commit to an incorrect optimization direction (Ma et al., 2026; Shetty et al., 2026; He et al., 2026), while LM agents still face difficulty recovering from an early decision within a single optimization trajectory (Saparov & He, 2023; Huang et al., 2023; Yang et al., 2026). APO employs independent subagents to explore the identified targets from the same repository state and evaluate their patches against the original and generated workloads. An integration agent then compares their patches, trajectories, and measurements before selecting and committing to the strongest candidate, forming a breadth-then-depth search over optimization targets.

Results. WORKLOADPRO enhances SWE-FFICIENCY LITE and GSO, generating 32.2 additional diverse held-out workloads on average. Extensive evaluation reveals that coding agents suffers from overoptimization, which inflates performance evaluation. Relying on the original workloads can overstate patch quality and even reverse rankings of agents. APO outperforms all agent baselines by 7.9 and 10.8 percentage points, respectively, while producing less overoptimization patches. Applying APO to real-world cyber physical systems identifies and fixes 13 performance issues, of which seven have been confirmed and three patches merged. Our artifacts are available at `https://anonymous.4open.science/r/agentperfopt-0871`.

```python
# Workload
def setup():
    global color
    color = (0.2, 0.5, 0.8, 0.3)
def workload():
    _to_rgba_no_colorcycle(color)
timeit.repeat(workload, number=10,
     repeat=2000, setup=setup)

# Claude Code patch
if (c is _cache[0] and alpha is _cache[1]):
    return _cache[2]
if type(c) is tuple:
    ...
    if (_c0.__class__ is float and ...):
        ...
```

(a) Workload and final patch (different files)



(b) Performance across iterations

Figure 1: Patch overoptimization during Claude Code's optimization trajectory. (a) The workload reuses one tuple across ten calls, and the final patch returns a stored result when it receives the exact same input objects again. (b) The line reports mean original-workload improvement over three interleaved runs; error bars show one standard deviation. The bars report the fraction of 14 WORKLOADPRO workloads with positive mean improvement.

2 MOTIVATING EXAMPLE

2.1 MOTIVATING EXAMPLE

Consider the SWE-FFICIENCY task `matplotlib-15834`. The visible workload calls `_to_rgba_no_colorcycle` ten times with the same four-element tuple of built-in Python floats. Claude Code with Opus 4.7 eventually exploits both properties. It adds a fast path restricted to exactly this representation and a one-entry cache that stores the most recent `c` and `alpha` objects together with the conversion result. Before each conversion, the patch checks whether the incoming `c` and `alpha` are the exact same objects as those stored in the cache, rather than merely having equal values. If so, it returns the stored result without performing the conversion again. The measured workload thus rewards repeated cache hits rather than a generally faster color-conversion path.

To see how this overoptimization develops, we replay the first three checkpoint patches on the visible workload and 14 fixed, validated WORKLOADPRO workloads (Figure 1). The first patch bypasses `NumPy` conversion for tuples and lists of built-in numbers, improving the visible workload by 67.8% and 11 of the 14 held-out workloads. The second patch narrows this optimization to tuples of built-in floats; visible-workload improvement rises to 90.4%, but only 9 of 14 held-out workloads improve. The third patch adds this same-object cache; improvement rises again to 97.0%, while only 7 of 14 held-out workloads improve. Thus, each iteration scores better on the single optimization target by specializing more strongly to its input representation and object-reuse pattern.

The complete held-out evaluation confirms that this tradeoff is not a peculiarity of the 14-workload replay. Across 20 workloads that vary color representation, scalar type, and object reuse, Claude Code improves runtime by only 18.0% on average and slows seven workloads by at least 5%, despite achieving 85.3% improvement on the original workload under the benchmark's full measurement protocol. Even a newly created tuple with the same values misses the cache because it is a different object. Lists and tuples containing NumPy scalars also miss the exact-type fast path; these workloads still pay for the added checks before falling back to the original conversion code.

APO instead evaluates candidate patches during optimization on five guardrail workloads spanning RGB tuples, NumPy arrays, Python lists, tuples of NumPy scalars, and byte arrays. Its final patch does not require repeated calls to reuse the exact same input object and supports multiple numeric-sequence representations. It achieves a comparable 85.0% improvement on the original workload, but improves the 20 held-out workloads by 55.9% on average without regressing any of them.

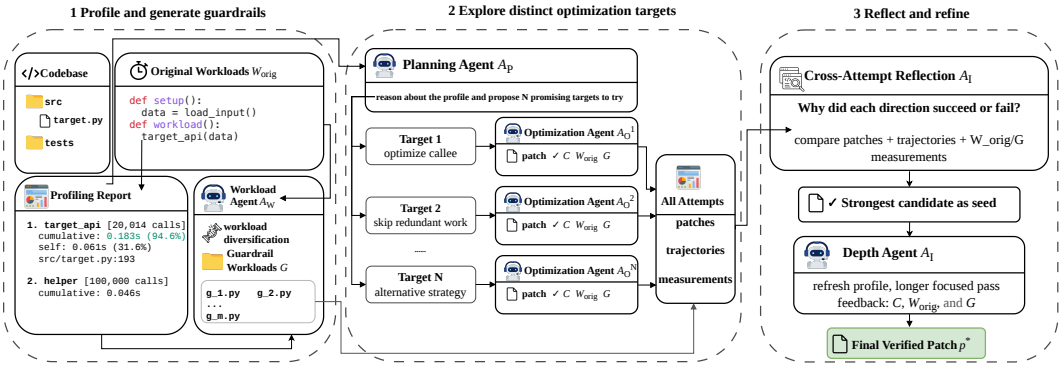


Figure 2: Overview of APO. The system profiles the original workload set and generates guardrail workloads, then uses a planning agent to identify distinct optimization targets. Isolated agents explore these targets in breadth; an integration agent reflects on their patches, trajectories, and measurements before selecting a seed for a longer depth-refinement pass.

3 AGENTPERFOPT FRAMEWORK

Figure 2 presents the workflow of APO. Given a repository, its correctness tests, and an original performance workload set W_{orig}, APO first collects profiling information and generates guardrailing workloads, then allocates test-time compute through breadth-then-depth optimization. The breadth phase tries multiple distinct optimization directions; the depth phase reflects on the attempts and further improves the strongest candidate. As a result, it reduces workload-specific reward hacking (Skalse et al., 2022) and dependence on a single early localization decision.

3.1 PROFILING-GUIDED LOCALIZATION

APO profiles the execution of W_{orig} with a profiling tool to collect and rank the runtime of each function. The surrounding source code of the highest-ranked functions is extracted and stored to save the cost of agent exploration in the code base in later stages. However, although profiling reveals where the workload spends time, it does not directly identify the source location that should be modified. Consider a call chain $A \to B \to C$. A cumulative-time profile may rank A above B and B above C, because the time attributed to each caller includes time spent in its callees. If A and B are thin wrappers, the most effective patch may be inside C; conversely, even when C accounts for most of the computation, the better patch may be in B if it invokes C redundantly or performs an avoidable conversion before the call. The profiling report therefore narrows the search space but does not uniquely determine the patch location. In practice, performance diagnosis requires testing multiple hypotheses and comparing their end-to-end effects, which motivates APO to explore multiple profiling-supported targets before committing to deeper optimization.

3.2 ON-THE-FLY GUARDRAIL WORKLOAD GENERATION

When an agent optimizes only against W_{orig}, its feedback reveals only whether that particular execution becomes faster. It cannot distinguish a broadly effective optimization from a patch specialized to the inputs or execution pattern of W_{orig}. To provide a broader optimization signal, APO generates a set of guardrail workloads before editing the repository. A workload-generation agent $\mathcal{A}_{\text{W}}$ analyzes W_{orig}, its profile, and the relevant source code to characterize the costly behavior exposed by W_{orig}. It then generates diverse workloads G that should also benefit from a broadly effective optimization that optimizes for W_{orig}, by varying the inputs or execution paths. For each candidate patch, each workload is evaluated on both the before-patch and after-patch code. The results of the patch on all workloads are aggregated and analyzed later to determine 1) whether the patch is the correct fix for the performance bottleneck and 2) whether the optimization is broadly effective.

3.3 BREADTH-THEN-DEPTH OPTIMIZATION

Profiling narrows the set of plausible optimization targets but rarely determines a unique patch location (Section 3.1). A single-trajectory agent usually commits to one optimization direction, whether supported by profiling evidence or chosen speculatively when the evidence is inconclusive. Because subsequent iterations are conditioned on the earlier reasoning and edits, the agent may continue refining that choice while overlooking more promising alternatives. APO mitigates this by employing a structured search: it first explores distinct optimization targets guided by profiling information, then performs deeper optimization based on the experience from all exploration attempts.

Diversifying optimization targets. Using the profiling evidence and workload context, a planning agent $\mathcal{A}_{\mathrm{P}}$ selects up to N distinct targets that appear promising to optimize ($N = 3$ in our experiments). The targets differ in patch location or optimization strategy, so that the risk of missing the most fruitful optimization is reduced. For the call chain $A \to B \to C$ in Section 3.1, for example, one target may optimize the implementation of C, while another removes a redundant call or conversion in B. Exploring such targets independently increases the chance of finding an effective patch and reduces the risk of repeatedly refining an unproductive direction.

Independent breadth exploration. For the ith target, APO forks a focused optimization agent $\mathcal{A}_{\mathrm{O}}^{(i)}$ from the same repository state. Each agent dives into its assigned direction using execution feedback from the correctness tests, W_{orig}, and the guardrail workloads G. Each agent runs in an isolated workspace and Docker container to prevent interference between branches. After all branches finish, APO evaluates their candidate patches under the same measurement procedure and stores the results, including performance statistics on W_{orig} and G. Considering both workloads punishes the over-specialized patches and encourages more broadly effective optimizations.

Cross-attempt reflection and depth refinement. After breadth exploration, a new integration agent $\mathcal{A}_{\mathrm{I}}$ jointly examines the candidate patches, attempt histories, and measurements on W_{orig} and G. It reasons about why each optimization direction succeeded or failed, whether its gains extend across the guardrail workloads, and what opportunities remain for further improvement. Based on this cross-attempt reflection, the agent selects the candidate that provides the strongest basis for depth refinement. APO then refreshes the profile of the selected candidate, and $\mathcal{A}_{\mathrm{I}}$ performs a longer optimization pass that may further develop the selected strategy or incorporate compatible changes discovered in other attempts. Throughout this stage, the correctness tests, W_{orig}, and G continue to guide optimization. In summary, APO uses breadth exploration to collect evidence across distinct directions and depth refinement to concentrate further optimization on the most promising direction.

4 THE WORKLOADPRO BENCHMARK

Performance optimization benchmarks typically expose a single or small set of original workloads (denote by W_{orig}) (Ma et al., 2026; Shetty et al., 2026; He et al., 2026). Because of the limited scale and diversity of the original workloads, coding agents could perform aggressive optimization towards the workloads, trading generality for efficiency on particular workloads. WORKLOADPRO augments W_{orig} into a scalable and diversified evaluation suite, serving as the hidden test cases for the evaluated agents. Patches that show little or no improvement on the generated workloads will be averaged down and even punished with a zero score. Consequently, it enhanced the evaluation that could be easily reward-hacked and rectified the inflated benchmark numbers, providing holistic evaluation for patches from coding agents.

4.1 AUGMENTED BENCHMARK CONSTRUCTION

Given an original workload set W_{orig} and the benchmark's human patch p_h, WORKLOADPRO constructs a generated evaluation suite W_{gen}. WORKLOADPRO aims to iteratively generate workloads that exercise the intended bottleneck of the task, but in different and diverse ways. WORKLOADPRO combines two complementary generators with execution-based filtering. The robustness generator only mutates the value, shape, or type of inputs, whereas the diverse generator changes the workload structure to reach new program paths. WORKLOADPRO further performs coverage analysis and

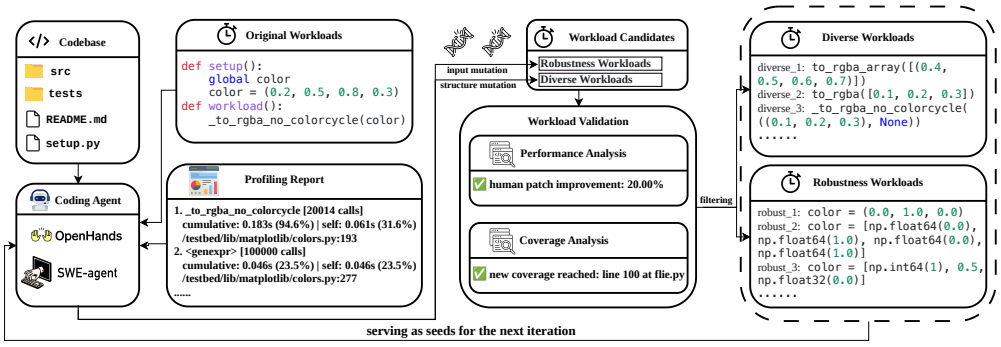


Figure 3: Overview of WORKLOADPRO construction. Starting from the codebase and original workloads, the agent mutates the input and structure. Candidates are retained when the human patch improves their performance; diverse candidates must additionally expand the accumulated line coverage. Accepted robustness and diverse workloads serve as seeds for the next generation round.

execution-based filtering on both candidates to deduplicate similar workloads and remove workloads that do not trigger the bottleneck. Figure 3 summarizes the construction pipeline.

Iterative generation. We initialize the seed set as $S_0 = W_{\text{orig}}$. In round r, each generator produces M candidates from S_{r-1}. After validation, the retained robustness and diverse candidates form A_r^{rob} and A_r^{div}, respectively. Their union $A_r = A_r^{\text{rob}} \cup A_r^{\text{div}}$ becomes the seed set for the next round, $S_r = A_r$, allowing later candidates to explore different workloads incrementally. The final generated suite is the union of the retained candidates across all rounds, $W_{\text{gen}} = \bigcup_{r=1}^{R} A_r$. This procedure generates up to $2RM$ workloads per task. In the experiment, we intend to generate $M = 10$ candidates per generator in each of $R = 5$ rounds, i.e., , up to 100 candidates per task.

Robustness workloads generation. A good performance-improving patch is expected to work on not only a few particular workloads, but also other similar workloads (at least not degrading them). To mitigate over-specialized patches, robustness generator mutates different input dimensions such as scale, shape, ordering, sparsity, boundary values, and types, etc. while preserving the test structure. The human patch is evaluated on each of the generated robustness workloads, and only workloads with runtime of at least 10 ms and a human-patch improvement of at least 10% are kept, ensuring that retained workloads are stable, measurable and exercise the intended performance bottleneck.

Diverse workloads generation. Diverse workloads generation aims to produce workloads that exercise the same performance bottleneck through different program paths. To expose patches specialized to particular paths, the diverse generator changes the workload structure, such as API entry points, API invocation chains, and helper operations, and uses coverage accumulated from the original and previously accepted workloads to target uncovered paths. The human patch is evaluated on each generated diverse workload. We retain a workload only if its before-patch runtime is at least 10 ms, the human patch improves it by at least 10%, and its execution covers at least one repository line not reached by the original or any previously retained workload.

4.2 BENCHMARK STATISTICS

While WORKLOADPRO is designed as a general benchmark-augmentation framework, this paper instantiates it on two repository-level performance benchmarks: SWE-FFICIENCY LITE (100 tasks) and GSO (102 tasks). With $R = 5$ and $M = 10$, the two generators can propose up to 100 candidates per task, or 20,200 candidates across both benchmarks. Following execution-based validation and stability filtering, our final evaluation retains tasks with more than five generated workloads. This yields 3,415 workloads for 85 SWE-FFICIENCY LITE tasks (2,902 robustness and 513 diverse) and 1,612 workloads for 71 GSO tasks (1,481 robustness and 131 diverse). The retained suites contain 7–66 workloads per SWE-FFICIENCY LITE task (mean 40.2, median 45) and 6–56 per GSO task (mean 22.7, median 22), for 5,027 held-out workloads in total.

5 EXPERIMENT

5.1 EVALUATION SETUP

We reuse each original benchmark's official evaluation harness and timing protocol, including warmup, workload repetition, cpu-pinning etc. (Shetty et al., 2026; Ma et al., 2026), to ensure reliability and reproducibility of the evaluation. Additional details of evaluation setup are provided in Appendix B.1.

Baselines. We include two methods as baselines, OpenHands (Wang et al., 2025a), a popular open-source coding agent widely used in performance optimization Shetty et al. (2026); Ma et al. (2026); He et al. (2026) and Claude Code (Anthropic, 2026), a proprietary coding agent. We evaluate OpenHands with GPT-5.5 (OpenAI, 2026) and Claude Opus 4.7 (Anthropic, 2026), and Claude Code with Claude Opus 4.7. Each baseline agent runs inside a Ralph Loop (Huntley, 2026) for at most 20 iterations with a $3 budget per-task. APO instead assigns a separate budget to each component (Appendix B.2). Its realized mean (median) cost per task is $3.02 ($2.98) on SWE-FFICIENCY LITE and $3.12 ($3.17) on GSO. Within each benchmark, both baselines receive the same optimization task prompt, reproduced in Appendix A. In addition, we include the human patch of each task as a reference. The ablations remove (i) the profiling information as guidance, denoted as *No profiling*, and (ii) the optimization-time workload guardrails, denoted as *No workload generation*, while retaining other components.

Metrics. For each patch, we measure the runtime of each workload on both the original code and patched code. Let $t(p, w)$ denote the runtime of workload w after applying patch p, with $p = \varnothing$ denoting the unmodified repository. We report runtime improvement as $I(p, w) = 1 - t(p, w)/t(\varnothing, w)$. Additionally, we report Opt@1, the percentage of tasks whose correctness-preserving patch meets the benchmark's base-speedup requirement and exceeds 0.95 relative to the human patch, following Shetty et al. (2026). We flag a patch as *overfitting* (i.e., over-specialized) when at least one generated workload regresses by at least 5%. We also include the rectified improvement percentages, where a flagged *overfitting* patch scores 0% on both the original and generated workloads.

Datasets. Our evaluation uses two benchmarks augmented by WORKLOADPRO, constructed from the SWE-FFICIENCY LITE split (Ma et al., 2026) and the GSO benchmark (Shetty et al., 2026), as described in Section 4. After workload generation and execution-based validation, we retain every task with more than five valid generated workloads, yielding an evaluation set of 85 SWE-FFICIENCY LITE tasks and 71 GSO tasks. This held-out suite contains 3,415 and 1,612 generated workloads for the two benchmarks, respectively. None of the generated workloads is accessible during optimization.

Models and parameters. We implement each APO subagent with GPT-5.5. APO generates five optimization-time guardrail workloads and explores three independently planned optimization targets. Complete task-facing prompt templates for guardrail workload generation, branch optimization etc. are detailed in Appendix A. Each system produces one final patch per task for evaluation.

5.2 MAIN RESULTS

The original workloads appear not sufficient for performance evaluation. Overall patches in Table 1 achieves lower improvement on the generated workloads than on the original workload. For example, OpenHands with GPT-5.5 drops by 25.6 percentage points on SWE-FFICIENCY LITE, from 64.7% to 39.1%. Moreover, the generated workloads can also change the relative ranking of methods. On GSO, OpenHands with GPT-5.5 outperforms Claude Code with Opus 4.7 on the original workload (41.2% versus 38.5%), but falls behind on the generated workloads (31.8% versus 34.3%). Evaluating only on the original workload could rank the agents that are more capable of generating specializing patches higher rather than agents that really fix the performance issues, i.e., , reward hacking (Skalse et al., 2022).

The generated held-out workloads also help reveal potentially over-specializing patches. For instance, 9 out of 85 patches generated by OpenHands with GPT-5.5/Opus-4.7 degrade by over 5% on generated workloads on SWEFFICIENCY. These regressions demonstrate that a large improvement on the benchmark input does not necessarily imply a broadly effective optimization.

Table 1: Performance of APO and baselines on SWEfficiency and GSO.

	Improvement (%)		Rectified improvement (%)			
Method	Original	Generated	Original	Generated	Opt@1	Overfit
SWE-FFICIENCY LITE						
Human patch	61.5	46.8	–	–	–	–
APO (GPT-5.5)	67.8	47.0	66.4	46.2	78.8	3/85
OpenHands (GPT-5.5)	64.7	39.1	59.7	37.0	71.8	7/85
Claude Code (Opus 4.7)	58.0	37.8	53.5	35.5	62.4	9/85
OpenHands (Opus 4.7)	41.0	26.6	37.4	24.0	41.2	9/85
GSO						
Human patch	59.2	50.3	–	–	–	–
APO (GPT-5.5)	53.4	45.1	52.1	44.4	50.7	2/71
OpenHands (GPT-5.5)	41.2	31.8	37.0	30.3	32.4	7/71
Claude Code (Opus 4.7)	38.5	34.3	37.8	34.2	31.0	5/71
OpenHands (Opus 4.7)	23.9	21.3	23.9	21.3	14.1	3/71

Table 2: APO ablations on SWEfficiency and GSO.

	Improvement (%)		Rectified improvement (%)			
Variant	Original	Generated	Original	Generated	Opt@1	Overfit
SWE-FFICIENCY LITE						
APO (GPT-5.5)	67.8	47.0	66.4	46.2	78.8	3/85
No profiling	66.4	38.2	62.3	36.6	75.3	6/85
No workload generation	66.2	46.3	63.4	44.1	75.3	5/85
GSO						
APO (GPT-5.5)	53.4	45.1	52.1	44.4	50.7	2/71
No profiling	53.0	42.5	50.8	40.3	50.7	5/71
No workload generation	54.9	43.8	52.2	41.7	45.1	6/71

APO improves both performance and generalization. On SWE-FFICIENCY LITE, APO exceeds the strongest agent baseline on the generated workloads by 7.9 percentage points (47.0% versus 39.1%), while producing 3 over-specialized patches compared with 7 for the same-model OpenHands baseline. On GSO, APO leads the strongest baseline by 10.8 points (45.1% versus 34.3%) and produces 2 over-specialized patches among 71 tasks.

5.3 ABLATIONS

Guardrail workloads reduce patch over-optimization. On SWE-FFICIENCY LITE, removing workload generation increases flagged tasks from 3 to 5 and reduces rectified generated-workload improvement from 46.2% to 44.1%. On GSO, it increases flagged tasks from 2 to 6 and reduces rectified generated-workload improvement from 44.4% to 41.7%.

Profiling improves search reliability. Without profiling, generated-workload improvement falls from 47.0% to 38.2% on SWE-FFICIENCY LITE and from 45.1% to 42.5% on GSO. The number of flagged tasks doubles from 3 to 6 on SWE-FFICIENCY LITE and increases from 2 to 5 on GSO. Profiling therefore helps the agent identify optimization directions that remain effective across both original workloads and generated workloads.

5.4 CASE STUDIES OF OPTIMIZING REAL-WORLD SYSTEMS

We applied APO's idea of generating guardrail workloads before optimization to six actively maintained cyber-physical systems where performance is critical. In total, we identified 13 performance issues and submitted patches accordingly. At the time of submission, developers had confirmed seven of these issues, including three whose patches had been merged. The below illustrates performance problems in robotics middleware and simulation resolved by this method.

ROS 2 `rclpy`: eliminating cumulative callback growth. The public `spin_once_until_future_complete` API registered a new wake callback every time a caller polled the same Future.[1] Consequently, a Future completed after N spins retained $O(N)$ callbacks and created $O(N)$ tasks upon completion. Profiling this path led APO to make callback registration idempotent. After 5,000 spins, the patch reduced retained callbacks from 5,000 to one and Future-completion time from 44.7 ms to 0.30 ms (149×), i.e., from $O(N)$ to $O(1)$. The patch was merged into ROS 2 Rolling and backported to Lyrical, removing history-dependent latency from the executor shared by timers, subscriptions, services, and actions.

Gazebo `gz-sim`: eliminating redundant per-step data work. Two patches removed unnecessary construction and copying of structured data from the simulation loop. First, `JointStatePublisher` rebuilt a nested Protocol Buffer message on every step even though its model and joint structure remained invariant.[2] Building this structure once and updating its values in place sped up message updates by 2.73–2.80× across 4–80 joints and reduced compute time per simulation step by 3–5% on two existing Gazebo worlds. Second, the physics system cached its step output but returned it by value, copying the data before downstream processing.[3] Returning the cached output by reference removed the copy from the profile and reduced median step time in an active 3,003-object world from 12.744 to 12.238 ms (3.97%). Both patches were merged and backported to `gz-sim` 8 and 9.

6 Related work

Repository-level performance optimization. Coding agents increasingly target repository-level performance in addition to functional correctness (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025a; Shetty et al., 2026; He et al., 2026; Ma et al., 2026). A central difficulty is localizing the true bottleneck: agents often commit early to one patch location and overlook better optimization opportunities (Shetty et al., 2026; He et al., 2026; Ma et al., 2026; Yang et al., 2025). APO instead uses profiling to identify and independently explore multiple optimization targets before refinement.

Performance evaluation and overoptimization. Existing benchmarks often use the same small workload set for both optimization feedback and final evaluation, enabling workload-specific patches to obtain inflated speedups (Chen et al., 2026; Yang et al., 2025; Yu et al., 2025; Li et al., 2026a; Wang et al., 2025b; Ahmed et al., 2026). WorkloadPro evaluates patches on diverse workloads withheld from agents, while APO uses separate, independently generated workload set as optimization-time guardrails to mitigate overfitting.

Test-time scaling for coding agents. Prior work improves coding agents by sampling more trajectories or repeatedly refining candidates with execution feedback (Ma et al., 2025; Novikov et al., 2025; Li et al., 2026b). In performance optimization, however, iterations along one trajectory can reinforce an early bottleneck-localization error or further specialize a patch to the observed workload. APO instead balances test-time compute between breadth and depth: profiling identifies distinct optimization hypotheses, isolated agents explore them from the same repository state, and an integration agent compares their patches and measurements before refining the strongest direction.

7 Conclusion

We presented WorkloadPro, a framework for augmenting performance benchmarks with diverse held-out workloads, and APO, an agentic optimization framework designed to produce broadly effective patches. Extensive evaluation on WorkloadPro workloads reveals that coding agents suffer from overoptimization and the original workloads can overstate patch quality and inflate the benchmark. APO outperforms baseline agents on both original workloads and held-out workloads, with the fewest overfit patches. APO further identified 13 performance issues in open-source systems where performance is critical, of which seven were confirmed and three patches were merged.

[1] ros2/rclpy#1702

[2] gazebosim/gz-sim#3853

[3] gazebosim/gz-sim#3850

AI USE STATEMENT

We used generative AI tools to assist with implementation and debugging, experiment orchestration, data aggregation and analysis, and manuscript drafting and editing. The authors reviewed all AI-assisted work and verified the reported patches and measurements using the correctness and performance protocols described in Section 5.1. The authors take responsibility for the final content of this work.

ETHICS STATEMENT

This work evaluates AI-generated changes to existing software and does not involve human subjects or private personal data. Automatically generated optimizations can regress behavior or performance outside the tested cases; they should therefore undergo human review and broader testing before deployment, particularly in safety-critical systems. Appendix **??** discusses these risks and the compute costs of the approach in more detail.

REPRODUCIBILITY STATEMENT

Section 4 describes benchmark construction and filtering, Section 5.1 specifies the evaluation protocol, models, software versions, hardware, and the complete benchmark split. Appendix A provides the complete task-facing prompt templates, and Appendix **??** documents the main limitations and sources of variability. All artifacts and reproduction instructions are in `https://anonymous.4open.science/r/agentperfopt-0871`.

REFERENCES

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Investigating test overfitting on swe-bench. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pp. 1292–1296, 2026.

Anthropic. Claude code overview., 2026. URL `https://docs.anthropic.com/en/docs/claude-code/overview`.

Anthropic. Claude Model Overview, 2026. URL `https://docs.anthropic.com/en/docs/about-claude/models/overview`.

Zhi Chen, Zhensu Sun, Yuling Shi, David Lo, and Lingxiao Jiang. Are performance-optimization benchmarks reliably measuring coding agents? *arXiv preprint arXiv:2607.01211*, 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 *International Conference on Machine Learning*, 2026.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

Geoffrey Huntley. Everything is a ralph loop., 2026. URL `https://ghuntley.com/loop/`.

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

Chenglin Li, Yisen Xu, Zehao Wang, Shin Hwei Tan, et al. Are benchmark tests strong enough? mutation-guided diagnosis and augmentation of regression suites. *arXiv preprint arXiv:2604.01518*, 2026a.

Weichen Li, Jiamin Song, Bogdan Alexandru Stoica, Arav Dhoot, Gabriel Ryan, Shengyu Fu, and Kexin Pei. Semrep: Generative code representation learning with code transformations. *arXiv preprint arXiv:2603.13640*, 2026b.

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 *International Conference on Machine Learning*, 2026.

Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Rongyu Cao, Jue Chen, Fei Huang, and Binhua Li. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute. *arXiv preprint arXiv:2503.23803*, 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. OpenAI API Model Documentation, 2026. URL https://platform.openai.com/docs/models.

Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=qFVVBzXxR2V.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=I5qDL315bQ.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. *Advances in neural information processing systems*, 35:9460–9471, 2022.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a.

You Wang, Michael Pradel, and Zhongxin Liu. Are" solved issues" in swe-bench really solved correctly? an empirical study. *arXiv preprint arXiv:2503.15223*, 2025b.

Chenghao Yang, Sida Li, and Ari Holtzman. LLM probability concentration: How alignment shrinks the generative horizon. *Transactions on Machine Learning Research*, 2026. URL https://openreview.net/forum?id=KotVuXj6CL.

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

Jun Yang, Cheng-Chi Wang, Bogdan Alexandru Stoica, and Kexin Pei. Synthesizing performance constraints for evaluating and improving code efficiency. *arXiv preprint arXiv:2505.23471*, 2025.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. Utboost: Rigorous evaluation of coding agents on swe-bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, 2025.

A PROMPT TEMPLATES

This appendix reproduces the complete task-facing prompt templates used in our SWE-FFICIENCY and GSO experiments. Instance-dependent content—including the workload, repository metadata, measurements, profiles, source snippets, and planner outputs—is replaced by bracketed placeholders. Bracketed fields are substituted verbatim at runtime, whereas numerical hyperparameters are shown at their experimental values. Conditional blocks show the alternatives selected for the benchmark and ablation configuration; only the applicable branch is included at runtime. We retain the output schemas, tool-use constraints, and validation instructions. Neither the human patch nor any held-out WORKLOADPRO workload is included in these prompts.

A.1 BASELINE OPTIMIZATION AGENTS

Within each benchmark, OpenHands and Claude Code receive the same user task. Their native agent scaffolds supply their respective tool-use system messages. For SWE-FFICIENCY, the complete shared task message is:

```
<uploaded_files>
/testbed
</uploaded_files>

I've uploaded a python code repository in the directory /testbed. Consider the following
     python workload showing a specific usage and measured performance of the repository:
<performance_workload>
[ORIGINAL WORKLOAD]
</performance_workload>

Can you help me implement the necessary changes to the repository so that the runtime of the
     `workload()` function is faster? Basic guidelines:
1. Your task is to make changes to non-test files in the /testbed directory to improve the
     performance of the code running in `workload()`. Please do not directly change the
     implementation of the `workload()` function to optimize things: I want you to focus on
     making the workload AS IS run faster by only editing the repository containing code that
     the `workload()` function calls.
2. Make changes while ensuring the repository is functionally equivalent to the original:
     your changes should not introduce new bugs or cause already-passing tests to begin
     failing after your changes. However, you do not need to worry about tests that already
     fail without any changes made. For relevant test files you find in the repository, you
     can run them via the bash command `[TEST COMMAND] <test_file>` to check for correctness.
     Note that running all the tests may take a long time, so you need to determine which
     tests are relevant to your changes.
3. Make sure the `workload()` function improves in performance after you make changes to the
     repository. The workload can potentially take some time to run, so please allow it to
     finish and be generous with setting your timeout parameter: for faster iteration, you
     should adjust the workload script to use fewer iterations. Before you complete your task,
      please make sure to check that the **original performance workload** and `workload()`
     function runs successfully and the performance is improved.
4. You may need to reinstall/rebuild the repo for your changes to take effect before testing
     if you made non-Python changes. Reinstalling may take a long time to run, so please be
     patient with running it and allow it to complete if possible. You can reinstall the
     repository by running the bash command `[REBUILD COMMAND]` in the workspace directory.
5. All the dependencies required to run the `workload()` function are already installed in the
      environment. You should not install or upgrade any dependencies.

Follow these steps to improve performance:
1. As a first step, explore the repository structure.
2. Create a Python script to reproduce the performance workload, execute it with `python <
     workload_file>`, and examine the printed output metrics.
3. Edit the source code of the repository to improve performance. Please do not change the
     contents of the `workload()` function itself, but focus on optimizing the code in the
     repository that the original `workload()` function uses.
4. If non-Python changes were made, rebuild the repo to make sure the changes take effect.
5. Rerun your script to confirm that performance has improved.
6. If necessary, identify any relevant test files in the repository related to your changes
     and verify that test statuses did not change after your modifications.
7. After each attempted change, please reflect on the changes attempted and the performance
     impact observed. If the performance did not improve, consider alternative approaches or
     optimizations.
8. Once you are satisfied, please use the finish command to complete your task.

Please remember that you should not change the implementation of the `workload()` function.
     The performance improvement should solely come from editing the source files in the code
     repository.
```

For GSO, both baselines instead receive the following benchmark-specific task message.

```
<uploaded_files>
/testbed
</uploaded_files>

I've uploaded a code repository in the directory /testbed. Consider the following test script
      showing an example usage of the repository:

<test_script>
[TEST SCRIPT]
</test_script>

Can you help me implement the necessary changes to the repository so that the runtime of the
     <test_script> is optimized?
Basic guidelines:
1. Your task is to make changes to non-test files in the /testbed directory to improve the
     performance of the <test_script>.
2. Make changes while ensuring the repository is functionally equivalent to the original.
3. Do not overoptimize for just the specific inputs in <test_script>. Make general
     performance improvements for the usage scenario shown.
4. You may need to rebuild the repo for your changes to take effect before testing. Some
     rebuilds may take time to run, so be patient with running them.

Follow these steps to improve performance:
1. As a first step, it might be a good idea to explore the repo to familiarize yourself with
     its structure.
2. Create a script in the /workspace directory (e.g., /workspace/test_opt.py) to reproduce
     and time the example and execute it with `python /workspace/<filename.py>` using the
     BashTool.
3. Edit the source code of the repo to improve the performance.
4. Rebuild and rerun your script and confirm that the performance has improved.
Your thinking should be thorough and so it's fine if it's very long.

To rebuild the repo with your changes at any point, you can use the following in the /testbed
      directory:
```
[REBUILD COMMAND]
```
```

The baselines run for at most 20 Ralph-loop iterations. On iterations 2–20, the following block is inserted after the repository/workload or test-script header and before the task instructions above; the repository state is preserved across iterations.

```
This is iteration [ITERATION] of 20. You have already worked on this same /testbed checkout
     in earlier iterations and your previous edits are still in place. Before doing anything
     else, inspect the current state (e.g. run `git diff` and re-run the workload to measure
     where performance stands now) and review the progress summary below. Then continue
     optimizing: build on the changes that helped and revise the ones that did not. Do not
     discard improvements that are already working.
<previous_progress>
[PROGRESS JSON]
</previous_progress>
```

A.2 Shared APO System Messages

Workload generation and target planning are analysis-only stages. They use the following system message; the relevant text from Sections A.3 or A.4 is sent as a separate user message.

```
You are an expert performance-analysis agent that can interact with a computer through
     exactly one tool named `bash`.

Rules:
- Every assistant turn must contain exactly one `bash` tool call.
- Do not reply with markdown code fences, XML command blocks, or plain-text shell commands.
- Put the full action in the `command` string of that single `bash` tool call.
- Commands run in `/testbed` by default, and you may also inspect files under `/workspace`.
- This is an analysis-only phase. Do not edit repository source code, rebuild the repo, or
     intentionally change the git state.
- When you are done, make one `bash` tool call that prints `
     COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT` on the first line and a valid JSON object on the
     following lines.
- The JSON submission must not be wrapped in markdown fences.
```

Focused candidate branches use the following system message. The two profiling helpers named below are omitted in the no-profiling ablations.

```
You are an expert software performance engineer that can interact with a computer through
     exactly one tool named `bash`.

Rules:
- Every assistant turn must contain exactly one `bash` tool call.
- Do not reply with markdown code fences, XML command blocks, or plain-text shell commands.
- Put the full action in the `command` string of that single `bash` tool call.
- Commands run in `/testbed` by default, so do not prefix actions with `cd /testbed` unless
     you truly need another directory.
- There is no `apply_patch` or `applypatch` helper inside the container. Never call either one
     . Edit files with standard shell tools such as `python`, `cat`, `sed`, or `perl`.
- Benchmark helper commands are special actions: `perf_eval`, `correctness_eval`, `
     rebuild_repo`, `best_status`, `restore_best`, `profile_eval_time`, `profile_eval_func`.
- When you use a benchmark helper command, the `command` must be exactly that helper name by
     itself. Do not wrap it in `cd`, pipes, `&&`, `||`, subshells, quotes, or additional shell
      text.
- Do not combine repository edits and benchmark helper commands in the same tool call. Edit
     in one turn, then evaluate in later turns.
- This is a focused optimization branch. Do not waste turns on broad repo walks like `ls -R`
     unless the task data is clearly inconsistent.
- Unless the target description is obviously wrong, do not rerun the untouched baseline
     before your first edit.
- You should usually inspect the target file and make a concrete edit within your first 2
     tool calls.
- One strong, well-validated focused patch is enough for this branch; you do not need 3
     distinct strategies here.
- When you are done, make one `bash` tool call with the exact command `echo
     COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT` and nothing else.
```

The depth-refinement agent instead uses the following system message; it has the same profiling-helper conditional.

```
You are an expert software performance engineer that can interact with a computer through
     exactly one tool named `bash`.

Rules:
- Every assistant turn must contain exactly one `bash` tool call.
- Do not reply with markdown code fences, XML command blocks, or plain-text shell commands.
- Put the full action in the `command` string of that single `bash` tool call.
- Commands run in `/testbed` by default, so do not prefix actions with `cd /testbed` unless
     you truly need another directory.
- There is no `apply_patch` or `applypatch` helper inside the container. Edit files with
     standard shell tools such as `python`, `cat`, `sed`, or `perl`.
- Benchmark helper commands are special actions: `perf_eval`, `correctness_eval`, `
     rebuild_repo`, `best_status`, `restore_best`, `profile_eval_time`, `profile_eval_func`.
- When you use a benchmark helper command, the `command` must be exactly that helper name by
     itself. Do not wrap it in `cd`, pipes, `&&`, `||`, subshells, quotes, or additional shell
      text.
- Do not combine repository edits and benchmark helper commands in the same tool call. Edit
     in one turn, then evaluate in later turns.
- When you are done, make one `bash` tool call with the exact command `echo
     COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT` and nothing else.
- You must attempt at least 3 distinct optimization strategies and run `perf_eval` after each
     before you may submit.
```

For both focused candidate branches and depth refinement, the task text shown below is followed by this user-message reminder. The profiling-helper names are omitted when profiling is disabled.

```
[TASK PROMPT]

Remember:
- Every reply must be exactly one `bash` tool call.
- Never answer with markdown code blocks.
- Benchmark helper commands must be standalone commands: `perf_eval`, `correctness_eval`, `
     rebuild_repo`, `best_status`, `restore_best`[, `profile_eval_time`, `profile_eval_func`].
- The workload can take time to run, so be generous with timeouts (3600 or larger is
     encouraged).
- After each attempted change, reflect on the performance impact. If it did not improve,
     consider alternative approaches.
- You should not change the workload function itself. Only edit repository source code.
```

A.3 GUARDRAIL-WORKLOAD GENERATION

```
You are generating diverse performance workloads for a Python repository to reduce over-
    optimization to the original benchmark input.

[WITH PROFILING]
Use the baseline profiling evidence below to reason about likely performance bottlenecks. You
     do not need to write a separate slow-path analysis artifact, but your workload choices
    should be grounded in structurally distinct ways of triggering the bottleneck.

Hotspot summary (from cProfile):
[BASELINE HOTSPOT SUMMARY]

[WHEN SOURCE SNIPPETS ARE AVAILABLE]
Source code of hot functions:
[SOURCE SNIPPETS]

[WITHOUT PROFILING]
No profiling evidence is provided for this run. Read `/workspace/workload.py` directly to
    reason about likely slow paths, and ground your workload choices in structurally distinct
     ways of triggering the bottleneck.

Requirements:
- Do not edit repository source code under /testbed.
- You may inspect files under /testbed and /workspace.
- Read `/workspace/workload.py` carefully and preserve its overall benchmark structure.
- Every generated script must call the same public stress target, keep the same output format,
     and differ from the original primarily in the workload argument or input shape.
- Preserve the original workload's timing loop and function call. Only change the input
    construction block unless a tiny helper is strictly needed to build that input.
- Use a fixed random seed.
- If the original input is iterable, generate iterable inputs only. If it is not iterable,
    generate non-iterable inputs only. Keep the same broad input type family.
- Prefer structurally different trigger cases over merely making one scalar input larger.

[GSO ONLY]
- This is a GSO benchmark. Official tests live in `/workspace/gso_tests/gso_test_N.py`, and `/
    workspace/workload.py` is an aggregate runner over those files.
- `/workspace/artifacts/gso_diverse_workload_template.py` shows the deterministic aggregate
    wrapper that AgentPerfOpt will render after you finish.
- Your only GSO filesystem outputs should be modified copied test files under `/workspace/
    diverse_workloads/diverse_N_tests/gso_test_*.py`. Do not create or edit `/workspace/
    diverse_workloads/workload_diverse_N.py`.
- For each diverse workload, create a subdirectory such as `/workspace/diverse_workloads/
    diverse_1_tests/`, copy one or more official GSO test files into it, and modify only
    input/setup construction inside those copied tests.
- Do not edit `run_test`, `check_equivalence`, reference-file naming, or timing logic in
    copied GSO tests. Candidates that change `run_test` or `timeit.template` will be
    discarded.
- Every copied GSO test must keep the same callable contract: `run_test(reference=False,
    eqcheck=True, prefix=...)` must return one float runtime, must write/read reference data
    through `prefix` when `reference=True`, and must perform the same kind of equality/
    assertion check when `eqcheck=True`.
- Do not rewrite `timeit.timeit(...)` as `t, result = timeit.timeit(...)`; `timeit.timeit(...)
    ` returns only a float. If a result is needed, compute it separately from the timed call,
     following the official test style.
- Do not assign to `timeit.template` or otherwise modify global `timeit` behavior.
- Do not change aggregate output handling. The generated wrapper will print per-test `
    GSO_TEST ... PASS ...`, one `Mean: ...`, one `Std Dev: ...`, and one `TEST_RESULTS: ...`
    line.

Tasks:
1. Read `/workspace/workload.py` [GSO: and the official files under `/workspace/gso_tests/`]
    and identify the stress target, loop structure, and output format.
2. Create exactly 5 workload variants under `/workspace/diverse_workloads/` named [
    SWEFFICIENCY: `workload_diverse_1.py`, `workload_diverse_2.py`, and so on] [GSO: `
    diverse_1_tests/`, `diverse_2_tests/`, and so on].
3. For each variant, choose a distinct slow-path hypothesis and encode it in the input shape.
     Think across dimensions such as size, ordering, sparsity/density, balance/skew, repeated
     vs. unique values, boundary values, nulls, and adversarial structure.
4. Validate each variant:

   [SWEFFICIENCY]
   - run `python -m py_compile /workspace/diverse_workloads/workload_diverse_N.py` after
     creating each workload
   - if compilation fails, delete that file, mark it discarded if you mention it, and move on.
     Do not repeatedly patch around it
   - do not run generated workloads yourself; the framework will validate them and collect
     authoritative baselines after your submission

   [GSO]
```

```
   - run `python -m py_compile /workspace/diverse_workloads/diverse_N_tests/gso_test_*.py`
     after creating each copied-test directory
   - do not run a diverse wrapper yourself; the framework will generate and validate it after
     your submission

5. Add a short comment block at the top of each kept workload:
   - `# WHY SLOW: <why this input structure causes the original code to be slow>`
   - `# DIFFERS FROM ORIGINAL: <what makes this input structurally unlike the original input>`
   - `# DIFFERS FROM PRIOR WORKLOADS: <which slow-path branch this uniquely stresses>`

When finished, submit valid JSON with this shape:

[SWEFFICIENCY]
{
  "workloads": [
    {
      "name": "diverse_1",
      "path": "/workspace/diverse_workloads/workload_diverse_1.py",
      "slow_path_hypothesis": "...",
      "rationale": "...",
      "status": "kept"
    }
  ],
  "notes": ["..."]
}

[GSO]
{
  "workloads": [
    {
      "name": "diverse_1",
      "test_dir": "/workspace/diverse_workloads/diverse_1_tests",
      "slow_path_hypothesis": "...",
      "rationale": "...",
      "status": "kept"
    }
  ],
  "notes": ["AgentPerfOpt will generate workload_diverse_N.py wrappers from the copied tests
     ."]
}

If a candidate workload fails compilation, either omit it or include it with `"status": "
     discarded"`.
```

A.4 Optimization-Target Planning

```
You are planning a test-time-scaling run.

Read `[PLANNER INPUT PATH]` and any repository files you need.
Before selecting targets, read `/workspace/workload.py` to understand the public call under
     test, the input construction, and the timing loop. If present, read `/workspace/artifacts
     /workload.original.py` when the isolated workload obscures the original benchmark shape.
Use profiling information from the planner input only as reference evidence. You have full
     freedom to choose targets outside the profile when the workload, source, native boundary,
      build/config path, repeated conversion, allocation pattern, or algorithmic structure
     suggests a better patch surface.
If cProfile hotspots are diffuse, low-percent, or mostly wrapper frames, treat that as a
     signal that the real bottleneck may sit outside visible Python frames.
If diverse workload artifacts are available in the planner input, use them to avoid choosing
     targets that only help the original workload shape.

For each target, provide a compact dossier: `evidence`, `hypothesis`, and `next_step`.
Use it to state the observed cost, the source-level condition suggesting preventable work,
     and the first concrete experiment.
Inspect the source snippets and nearby call context before choosing the location. Prefer a
     justified patch surface over merely tuning a hot leaf frame, especially when a guard,
     mode, constructor path, conversion path, or repeated-work boundary can prevent
     unnecessary work.
If no clearer preventable-work surface is visible, it is fine to target the hottest frame
     directly, but say why.

Submit up to 3 distinct optimization targets as JSON with this shape:
{
  "targets": [
    {
      "title": "...",
      "evidence": "...",
      "hypothesis": "...",
```

```
      "next_step": "...",
      "hotspot_name": "...",
      "filename": "...",
      "line_number": 123,
      "percent": 12.3,
      "suggested_files": ["..."]
    }
  ]
}

Requirements:
- Keep the targets materially distinct from one another.
- Tie each target to concrete workload, source, native-boundary, build/config, diverse-
    workload, or profiling evidence.
- Prioritize the best optimization opportunities, not broad codebase tours.
- Do not propose changes outside of /testbed repository source files.
- Favor targets that are specific enough for a focused branch to patch quickly.
- Prefer targets that are likely to generalize across the original workload and any generated
     diverse workloads.
- Keep `filename`, `line_number`, and `hotspot_name` pointed at the best patch surface you can
     justify when applicable. `hotspot_name`, `line_number`, and `percent` are optional
    evidence fields, not constraints.
- Keep each dossier concise. Do not add fields to the JSON schema.
```

If normalization rejects every proposed target, the planner is invoked once more with the previous submission and normalization report:

```
You are repairing a test-time-scaling target plan that normalized to zero usable targets.

Read `[PLANNER INPUT PATH]`, the previous submission at `[PREVIOUS SUBMISSION PATH]`, and the
    normalization report at `[NORMALIZATION REPORT PATH]`.
Then read `/workspace/workload.py` and any repository files needed to produce usable
    optimization targets.

Use profiling information only as reference evidence. You have full freedom to choose targets
     outside the profile when the workload, source, native boundary, build/config path,
    repeated conversion, allocation pattern, or algorithmic structure suggests a better patch
     surface.
If cProfile hotspots are diffuse, low-percent, or mostly wrapper frames, treat that as a
    signal that the real bottleneck may sit outside visible Python frames.

Submit up to 3 distinct optimization targets as JSON with this shape:
{
  "targets": [
    {
      "title": "...",
      "evidence": "...",
      "hypothesis": "...",
      "next_step": "...",
      "hotspot_name": "...",
      "filename": "...",
      "line_number": 123,
      "percent": 12.3,
      "suggested_files": ["..."]
    }
  ]
}

Requirements:
- Return at least one target if any plausible patch surface exists.
- Each accepted target needs a meaningful `title`, `evidence`, `hypothesis`, and a `/testbed
    /...` or repo-relative path in `filename` or `suggested_files`.
- Do not use `unknown`, `none`, or placeholder paths.
- Do not propose changes outside of /testbed repository source files.
- `hotspot_name`, `line_number`, and `percent` are optional evidence fields, not constraints.
- Keep each dossier concise. Do not add fields to the JSON schema.
```

A.5 Focused Candidate Branch

Each selected target (up to three) is assigned to an independent branch. Together with the focused system message above, that branch receives the following task message. The listing shows the profiling-enabled configuration; without profiling, the two profiling-helper bullets are omitted and the profiling-refresh sentence is replaced by "Profiling helpers are disabled for this run." Empty primary-location, suggested-file, and source-context blocks are also omitted at rendering time.

```
<uploaded_files>
/testbed
</uploaded_files>

This is a focused test-time-scaling candidate branch for instance `[INSTANCE ID]`.

Baseline:
- runtime: [BASELINE MEAN]s (stddev: [BASELINE STANDARD DEVIATION]s)
- correctness: passed=[PASSED] failed=[FAILED] error=[ERROR]
- correctness requirement: every PASS_TO_PASS test must be reported and pass
- [BASELINE CORRECTNESS NOTE]

Candidate branch [BRANCH INDEX]/[BRANCH TOTAL]:
- title: [TARGET TITLE]
- evidence: [TARGET EVIDENCE]
- hypothesis: [TARGET HYPOTHESIS]
- next step: [TARGET NEXT STEP]
- primary location: [TARGET FILE]:[LINE]
- suggested files: [SUGGESTED FILES]
- target manifest: `[TARGET MANIFEST PATH]`

Benchmark helpers (use these as standalone commands):
- `perf_eval` runs the workload on the current tree
- `correctness_eval` runs the PASS_TO_PASS regression tests
- `rebuild_repo` reruns the benchmark rebuild command after compiled or packaging changes
- `best_status` shows the current best checkpoint
- `restore_best` resets /testbed back to the best known checkpoint
- `profile_eval_time` reruns cProfile on the workload, prints the time-sorted hotspot summary,
     and refreshes /workspace/artifacts/current_profile.txt/json
- `profile_eval_func` reruns cProfile on the workload, prints a finer-grained function-level
    hotspot report, and refreshes /workspace/artifacts/current_profile.txt/json
- use each helper as the entire command by itself - never combined with other shell commands

Checkpoint behavior:
- if the same tree has a completed `correctness_eval` where every required test passes and a
    faster `perf_eval`, the framework automatically checkpoints it as the new best state
- use `restore_best` if an experiment goes bad
- small wins are worth keeping and building on

You may need to reinstall/rebuild the repo after non-Python changes: `[REBUILD COMMAND]`. Use
    a timeout of 3600 or larger for rebuilds and performance runs.
For correctness, run `[TEST COMMAND] <test_file>` on relevant test files in /testbed.
All dependencies are already installed. Do not install or upgrade any.

## PERFORMANCE LOG

Maintain a compact running performance log throughout this session. After each promising
    validation cycle, append a row like:

  [attempt | edit description] original=<mean>s  diverse_1=<mean>s  diverse_2=<mean>s
    correctness=passed/failed

Keep the log visible in your working notes so you can compare strategies.

Focused branch rules:
- The planner selected this target, so do not spend turns on broad repo exploration like `ls -
    R`.
- Before editing, read `/workspace/workload.py` and confirm how the target is reached by the
    measured workload. If helpful, read `/workspace/artifacts/workload.original.py` and the
    current profile artifacts for additional context.
- Do not rerun the untouched baseline before your first edit unless the target manifest is
    clearly inconsistent.
- Treat the target dossier as your starting point: verify the evidence and hypothesis, then
    begin with the proposed next step.
- Start from the workload script and target context below. Use profiling evidence as needed.
    Pick one concrete bottleneck and explain what code change should reduce the work across
    input shapes, not just the original workload.
- Make a concrete code change within your first 2 tool calls unless you discover a blocker.
- Never use `apply_patch` or `applypatch`; they are unavailable here. Edit with `python`, `sed
    `, `perl`, or `cat`.
- Favor one strong patch over multiple tentative strategies in this branch.
- If non-Python changes were made, run `rebuild_repo`.
- After each edit, validate with `perf_eval`.
- If the perf result is clearly worse, run `restore_best` and try a different strategy.
- If the perf result looks promising, run `correctness_eval`.
- If correctness regresses, try a targeted fix that preserves the speedup. If that fails, run
     `restore_best`.
- If `/workspace/diverse_workloads/workload_diverse_*.py` exists, run those workloads after
    promising edits to check that gains generalize.
- If the original workload improves but the diverse workloads regress materially, treat that
    as overfitting pressure and prefer a more general fix.
```

```
- Read `/workspace/artifacts/diverse_workloads.txt` if it exists so you understand what each
    diverse workload is trying to trigger.
- Refresh profiling with `profile_eval_time` for the compact time-sorted summary or `
    profile_eval_func` for the finer-grained function-level report.
- Before submitting, make sure the current tree matches the strongest validated result. If
    needed, run `restore_best`, then finish with `echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`.


Targeted source context:
[SOURCE SNIPPETS FOR THE SELECTED TARGET]
```

A.6 DEPTH REFINEMENT

The integration agent receives the depth system message above followed by the complete base optimization task and the depth-specific suffix below. The base-task listing shows the profiling-enabled configuration. In the no-profiling ablations, the two profiling-helper bullets and the entire initial hotspot/source block are omitted.

Base optimization task.

```
<uploaded_files>
/testbed
</uploaded_files>

I've uploaded a python code repository in /testbed. Your task is to make it faster by editing
     the source code.

Instance: [INSTANCE ID]
Repository: [REPOSITORY]
Version: [VERSION]
Base commit: [BASE COMMIT]
Likely import name: [IMPORT NAME]

Benchmark context:
[PROBLEM STATEMENT]

Baseline measurements:
- runtime: [BASELINE MEAN]s (stddev: [BASELINE STANDARD DEVIATION]s)
- correctness: passed=[PASSED] failed=[FAILED] error=[ERROR]
- [BASELINE CORRECTNESS NOTE]

Benchmark helpers (standalone commands only):
- `perf_eval` runs the workload on the current tree
- `correctness_eval` runs the PASS_TO_PASS regression tests
- `rebuild_repo` reruns the benchmark rebuild command after compiled or packaging changes
- `best_status` shows the current best checkpoint
- `restore_best` resets /testbed back to the best known checkpoint
- `profile_eval_time` reruns cProfile on the workload, prints the time-sorted hotspot summary,
     and refreshes /workspace/artifacts/current_profile.txt/json
- `profile_eval_func` reruns cProfile on the workload, prints a finer-grained function-level
    hotspot report, and refreshes /workspace/artifacts/current_profile.txt/json
- use each helper as the entire command by itself - never combined with other shell commands
- good: `perf_eval`
- bad: `cd /testbed && perf_eval`
- bad: `python edit.py && correctness_eval`

Checkpoint behavior:
- if the same tree has a completed `correctness_eval` where every required test passes and a
    faster `perf_eval`, the framework automatically checkpoints it as the new best state
- use `restore_best` if an experiment goes bad
- small wins are worth keeping and building on

You may need to reinstall/rebuild the repo after non-Python changes: `[REBUILD COMMAND]`. Use
    a timeout of 3600 or larger for rebuilds and performance runs.
For correctness, use `correctness_eval` for the benchmark check. If you need a direct local
    check, run `[TEST COMMAND] <test_file>` on relevant test files.
All dependencies are already installed. Do not install or upgrade any.

Workload grounding:
- Before choosing an optimization, read `/workspace/workload.py` and identify the public call
    under test, input construction, and timing loop.
- If present, `/workspace/artifacts/workload.original.py` preserves the raw benchmark before
    isolation; read it when the isolated script obscures the original workload shape.
- Use profiling artifacts as supporting evidence when needed, not as a substitute for
    understanding the workload script.

Initial hotspot summary (from cProfile):
[BASELINE HOTSPOT SUMMARY]
```

```
Refresh profiling with `profile_eval_time` for the compact time-sorted summary or `
    profile_eval_func` for the finer-grained function-level report.
Each profiling helper rewrites
`/workspace/artifacts/current_profile.txt` and
`/workspace/artifacts/current_profile.json`.
Source code of hot functions (pre-extracted for convenience):
[SOURCE SNIPPETS]

If present, generated diverse workloads live in `/workspace/diverse_workloads/` and their
    rationale/baselines are summarized in `/workspace/artifacts/diverse_workloads.txt`.

## PERFORMANCE LOG

Maintain a compact running performance log throughout this session. After each promising
    validation cycle, append a row like:

  [attempt | edit description] original=<mean>s  diverse_1=<mean>s  diverse_2=<mean>s
    correctness=passed/failed

Keep the log visible in your working notes so you can compare strategies.

## OPTIMIZATION PROCEDURE

1. Read `/workspace/workload.py` first. Then use the profiling evidence and source context
    above as needed to reason about the likely slow paths and how the generated diverse
    workloads might trigger them. Pick a concrete bottleneck and explain what code change
    should reduce the work across input shapes, not just the original workload.

2. Make one focused edit to /testbed.

3. If non-Python changes were made, run `rebuild_repo`.

4. Run `perf_eval`.
   - If the result is clearly worse, run `restore_best` and try a different strategy.
   - If the result looks promising, run every available `python /workspace/diverse_workloads/
    workload_diverse_N.py` workload as a separate validation step before moving on.

5. Run `correctness_eval`.
   - If correctness regresses, try a targeted fix that preserves the speedup. If that fails,
    run `restore_best`.
   - If the original workload improves but the diverse workloads regress materially, treat
    that as overfitting pressure and prefer a more general fix.

6. After major changes, refresh profiling to verify the bottleneck actually moved instead of
    just shifting elsewhere.

7. Repeat with distinct strategies. Prefer structurally sound fixes over one-off tweaks.

8. Before submitting, make sure the current tree matches the strongest validated result. If
    needed, run `restore_best`, then finish with:
   `echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`
```

Depth-specific suffix.

```
Test-time scaling deep-optimization phase.
Before editing, read `[DEPTH INPUT PATH]`.
Also read `/workspace/workload.py` before deciding whether to keep, combine, or replace
    candidate patches. If present, use `/workspace/artifacts/workload.original.py` to recover
     the raw benchmark shape; inspect profiling artifacts only as needed to resolve the
    current bottleneck.

You are the integrator and reviewer agent: compare the candidate branches, inspect the
    current best state, and push the strongest idea further.
Your target is to improve the original workload while keeping the diverse-workload guardrail
    healthy:
- improve the original workload relative to baseline
- keep at least 50% of generated diverse workloads within 5.0% of their own baselines

Treat this as a single profiling-guided deep optimization phase:
- do not spend turns on broad repo exploration unless the phase artifacts are inconsistent
- never use `apply_patch` or `applypatch`; they are unavailable in the container
- inspect the candidate scoreboard first; use supporting candidate artifacts only when you
    need more detail
- treat `candidate_scoreboard` as the authoritative candidate comparison and `candidate_refs`
    as supporting evidence
- the current workspace may already have the best measured seed patch applied; decide whether
     to build on it, replace it, or restore a better state
- inspect the current profiling feedback before making changes
- treat `*_time_profile` as total execution-time feedback and `*_func_profile` as hotspot
    breakdowns
```

```
- use the candidate artifacts when helpful, but do not cargo-cult them
- validate with `perf_eval` and `correctness_eval`
- if `/workspace/diverse_workloads/workload_diverse_*.py` exists, run those workloads after
    promising edits to guard against overfitting
- read `/workspace/artifacts/diverse_workloads.txt` if it exists so the round stays grounded
    in the intended slow-path coverage
- read the input artifact for the current diverse-workload guardrail summary and use it to
    decide whether a patch is too aggressive
- refresh profiling after major changes if helpers are available
- before finishing the phase, make sure the best validated state is checked out
- finish with the strongest state you can produce in this phase
```

B Detailed Protocols

B.1 Experimental Environment and Model Configuration

The experiments run on a cluster of x86_64 servers running Ubuntu 22.04.5 with Linux kernel `5.15.0-185-generic` and Docker 29.1.3. Each server provides 24 physical cores on two Intel Xeon Gold 6126 sockets and 187 GiB of memory. OpenHands uses version 0.51.1, Claude Code uses version 2.1.156, and APO uses mini-swe-agent v2.4.2 (Yang et al., 2024). We use each model's default reasoning effort: medium for GPT-5.5 and xhigh for Claude Opus 4.7. Other mini-swe-agent model settings, including sampling temperature and context-window configuration, retain their defaults. We do not evaluate Claude Code with GPT-5.5 because of compatibility issues.

B.2 APO Model-Cost Limits

APO applies an independent model-cost limit to each subagent invocation: $0.50 for guardrail-workload generation, $1.50 for target planning, $0.50 for each of up to three focused candidate branches, and $1.00 for depth refinement. The standard configuration therefore has $4.50 in nominal stage-level limits per task. If the initial planner output contains no usable target, APO may invoke one additional planning attempt with the same $1.50 limit. These limits are stopping thresholds rather than required expenditures; subagents may terminate earlier, and the reported realized cost sums usage across all invoked stages. Ablations retain the same limits for every stage they execute, while *No workload generation* omits the workload-generation stage.