Are Performance-Optimization Benchmarks Reliably Measuring Coding Agents?

Zhi Chen
Singapore Management University
Singapore, Singapore
zhi.chen.2023@smu.edu.sg

Zhensu Sun*
Singapore Management University
Singapore, Singapore
zssun@smu.edu.sg

Yuling Shi
Shanghai Jiao Tong University
Shanghai, China
yuling.shi@sjtu.edu.cn

David Lo
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

Lingxiao Jiang
Singapore Management University
Singapore, Singapore
lxjiang@smu.edu.sg

***Abstract*—Repository-level performance-optimization benchmarks such as GSO, SWE-Perf, and SWE-fficiency evaluate coding agents by applying patches to real repositories and comparing runtime against unoptimized baselines and official reference patches. Their leaderboard scores are increasingly used as evidence of coding-agent progress, but those scores can conflate runtime instability, benchmark-specific scoring rules, and how many tasks are already solved by at least one public submission. We audit these issues across the three benchmarks. First, we replay the official reference patches for 740 code optimization tasks across four common types of Google Cloud machines. Most benchmark tasks can be replayed, but their reference patches satisfy the original benchmark validity rules in every cross-machine replay for only 39/102 GSO tasks, 11/140 SWE-Perf tasks, and 411/498 SWE-fficiency tasks; SWE-Perf is especially fragile because many reference patches produce close-to-zero runtime changes. Second, we show that public submission rankings depend strongly on the benchmark scoring rule. Among eight public submissions shared by GSO and SWE-fficiency, the official rankings disagree on 9 of 28 pairwise submission comparisons, and SWE-fficiency's leaderboard scoring rule assigns the worst ten tasks overly high score weights of 58.5%–82.8%. Third, looking across 10 public submissions for each task, we find that at least one submission matches or beats the reference patch on 85.3% (384/450) of replay-valid GSO and SWE-fficiency tasks, and beats the unoptimized base code on 99.8% (449/450). Our study complements leaderboard scores by identifying tasks with more reliable performance signals, quantifying per-task score contributions, and exposing the remaining performance gaps that are hidden by aggregate rankings.**

***Index Terms*—software engineering benchmarks, performance optimization, coding agents, benchmark validity**

I. Introduction

Improving software performance is a practical and commercially important software-engineering task. Given a codebase and a workload, a coding agent is expected to produce a patch that preserves correctness while improving the program's performance [1]–[3]. Code-efficiency benchmarks have also grown in scope. Earlier benchmarks focus on function-level edits [4], standalone programs [5], [6], and function-level and file-level code generation tasks [7]–[9]. Recent repository-level benchmarks such as GSO [10], SWE-Perf [11], and SWE-fficiency [12] require agents to edit real projects and evaluate those edits with executable workloads, reference patches, and benchmark-specific scoring rules.

Such performance benchmarks differ significantly from functional repair benchmarks because they measure a non-functional property. Functional repair benchmarks such as SWE-Bench mainly judge whether the relevant tests pass, a binary and usually reproducible outcome [13]. Performance benchmarks must run the base program, the reference patch, and submitted patches under workloads, then compare noisy runtime measurements. These measurements are not fixed quantities: they fluctuate across runs because of CPU scheduling, cache state, memory bandwidth contention, and machine-level microarchitectural effects [14]–[16]. The same patch can therefore behave faster, slower, or statistically unsupported depending on where and how it is replayed, which brings instability to the benchmarks' final evaluation.

Recent repository-level benchmarks address this problem with repeated trials, outlier filtering, statistical tests, reference patches, and workload-selection rules. However, a fundamental question remains: to what extent can we trust their evaluation results, even with these countermeasures in place? We therefore audit three recent repository-level performance-optimization benchmarks: GSO [10], SWE-Perf [11], and SWE-fficiency [12]. We ask whether official reference patches remain valid under cross-machine replay, how scoring rules shape leaderboard rankings, and whether replay-valid tasks still expose clear performance gaps for recent top submissions. Notably, these public leaderboards consist of OpenHands-based submissions, each pairing the agent scaffold with one underlying foundation model [17]–[19]. In the remaining text of this paper, we use *submission* for the evaluated agent configuration and use model names only as shorthand labels for the underlying model component [20]–[22].

Our systematic assessment yields three main insights. First, the validity of reference patches is less reliable than what may be assumed by benchmark users. We replay every official reference patch across four Google Cloud machine types

*Corresponding author.

and three rounds, while preserving each benchmark's official workloads and performance validity rules. Most tasks remain executable, but only $39/102$ GSO tasks, $11/140$ SWE-Perf tasks, and $411/498$ SWE-fficiency tasks satisfy their original validity rules in all cross-machine replays. SWE-Perf is especially fragile because many reference patches produce only close-to-zero runtime changes. This means that some official tasks lose their performance signals when replayed on different machines, even if the task remains executable.

Second, benchmark scoring rules can strongly shape submission rankings. Among eight public submissions shared by GSO and SWE-fficiency, the official leaderboards disagree on 9 of the 28 pairwise submission orders. Part of this disagreement comes from the scoring rule itself: SWE-fficiency's leaderboard scoring rule gives very large weight to low-speedup tasks, so the worst ten tasks can carry 58.5–82.8% of a submission's score weight. A simple bounded-penalty diagnostic changes 6/8 submission ranks and flips 8/28 head-to-head comparisons. The ranking is therefore not only a submission comparison; it also reflects the penalty budget built into the scoring rule.

Third, because current agent systems often use multi-agent workflows rather than a single run, we ask an optimistic task-level question: if we look across 10 public submissions for each replay-valid task, which tasks still fall short of the reference patch? This separates tasks already covered by at least one public submission from tasks that still expose performance-optimization gaps. Across 450 replay-valid GSO and SWE-fficiency tasks, at least one public submission matches or beats the reference patch on 384 tasks; all 450 have a passing public patch; and 449 have a patch that beats the base program. The remaining 66 tasks are therefore rarely basic correctness failures. Most already have a useful public optimization, but that optimization does not yet reach reference-patch speed. The remaining gap is thus less about finding any working optimization and more about reaching or exceeding the reference-level target. As public submissions improve, future benchmarks may need stronger reference targets to keep separating stronger submissions from weaker ones.

Together, these results imply that leaderboard scores alone are not enough to interpret the progress of performance-optimization agents. Readers need to know which tasks have stable reference signals, how much the scoring rule amplifies extreme cases, and how the remaining unsolved tasks affect the submissions' scores.

Our contributions in this paper are as follows:

- We replay 740 official reference patches across four cloud machine types and twelve machine-round combinations, showing where each benchmark's reference signal remains stable and where it does not.
- We audit leaderboard rankings with released public outputs, showing that scoring rules can materially change submission rankings and that SWE-fficiency's leaderboard scoring rule is especially sensitive to a few low-performing tasks.
- We inspect task-level outputs across public submissions, showing that most already have passing, faster-than-base public patches and that the remaining gap is mainly about reaching reference-patch speed.

The rest of the paper is organized as follows. Section II defines the benchmark selection and research questions. Section III studies cross-machine reference-patch validity. Section IV audits scoring-rule sensitivity. Section V asks how many replay-valid tasks are covered when we look across 10 public submissions for each task. Section VI discusses implications for benchmark users, agent builders, and future benchmark design.

II. Study Design

A. Benchmark Selection

We study three repository-level benchmarks that directly target coding agents on performance optimization. Note that we are not surveying every efficiency benchmark for code models, nor every open-source benchmark artifact. We include benchmarks only when they (1) evaluate repository-level edits, (2) target runtime optimization with executable workloads or tests that compare code states, and (3) appear as recent, visible peer-reviewed benchmark papers rather than leaderboard-only, code-only, or unpublished proposals. These criteria select GSO, SWE-Perf, and SWE-fficiency because they combine executable tasks with reviewed task definitions and are likely to be reused to evaluate claims about agents' performance-engineering ability.

TABLE I
Repository-level performance-optimization benchmarks.

Benchmark	Scale	Runtime performance comparison tests
GSO *NeurIPS 2025*	102 tasks from 10 repos	Generated performance tests selected to compare base and reference optimization commits [10].
SWE-Perf *ICML 2026*	140 tasks from 9 repos	Unit tests filtered from source repositories to compare original and PR-modified commits [11].
SWE-fficiency *ICML 2026*	498 tasks from 9 repos	Annotated workload scripts compare pre-edit and expert-optimized commits [12].

B. Research Questions

RQ1: Do official reference patches remain valid under cross-machine replay?

Motivation: Code performance-optimization benchmarks are machine-agnostic: their tasks ask agents to optimize a program, without targeting a particular CPU model or machine configuration. This setup assumes that each benchmark's reference patch, together with the workload used to distinguish patches, produces a performance signal that survives reasonable machine variation. Cross-machine replay is therefore important: if the reference optimization is not stable across machines, later submission comparisons on the same benchmark rest on an unstable target. The replay checks whether each task remains evaluable, faster than base, and valid by its original benchmark rule.

RQ2: How do leaderboard scoring rules shape submission rankings?

Motivation: A benchmark compresses hundreds of task outcomes into one submission score. This compression decides

which submission appears stronger. In performance optimization, the same patches can look different depending on whether the rule counts only patches that match or beat the benchmark reference patch, gives partial credit for smaller speedups, or heavily penalizes very slow tasks. If a submission's rank depends on these scoring choices, then the leaderboard ranking alone does not tell readers what kind of progress the submission made. It is therefore necessary to make the scoring rules explicit, compare shared public submissions under them, and audit whether rankings are driven by broad task performance or by a small set of high-leverage scoring cases.

RQ3: Do benchmark tasks remain hard for top submissions?

Motivation: In practice, current agent systems often use multi-agent workflows rather than relying on a single run [23], [24]. However, the public leaderboards we analyze rank individual OpenHands-based submissions, each pairing the agent scaffold with one underlying model [2]. This mismatch matters because different submissions may solve different tasks, so a single leaderboard entry should not be treated as the boundary of task solvability. We therefore inspect task-level outputs across public submissions to estimate how many tasks are covered by at least one submission and whether the remaining gaps come from correctness, speedup depth, or optimization strategy.

III. RQ1: Cross-Machine Reference-Patch Validity

We replay every official reference patch on four Google Cloud machine profiles and reapply each benchmark's original validity rule. The goal is not to invent a stricter benchmark, but to test whether the original reference signal survives reasonable cross-machine variation.

A. Experiment Design

Machine selection. To test whether official reference patches remain faster than the unoptimized base under common machine differences, we use four Google Cloud machine profiles[1] with the same resource configuration: 64 vCPUs and 256GB of memory. This resource configuration also matches the Google Cloud `n2-standard-64` setup reported by GSO and SWE-fficiency in their original benchmark experiments [10], [12]. We therefore anchor the replay campaign in the same cloud provider and resource configuration, then vary the processor platform around that baseline. The profiles cover the two main server CPU vendors, Intel and AMD, and include both older and newer Google Cloud processor generations. This keeps the cloud provider and resource configuration fixed while varying the kind of hardware difference that benchmark users are likely to encounter in practice (Table II).

Experiment setup. We follow each benchmark's guidance and reuse the official tasks, reference patches, selected tests, and workload definitions. For each benchmark, we use the latest public repository version available before our April 30, 2026 data-collection cutoff.

Evaluation metrics. We report two task-level metrics because reference stability has two layers. *Faster-than-base* checks the minimum performance direction: the reference patch must beat the base program in every replay round and machine, $T_{ref,r,m} < T_{base,r,m}$. *Original-rule valid* checks the benchmark's own construction standard: the replay must still satisfy the rule that originally qualified the task as a performance-optimization instance. Table III lists those original rules from the benchmark papers [10]–[12].

[1]https://docs.cloud.google.com/compute/docs/general-purpose-machines

TABLE II
REPLAY MACHINE PROFILES.

Google Compute Engine	Processor platform	Google Cloud release year
`n2-standard-64`	Intel Cascade Lake, 2nd Gen Xeon, 2.80GHz	2019
`n2d-standard-64`	AMD Milan, 3rd Gen EPYC, 7B13	2021
`n4-standard-64`	Intel Emerald Rapids, 5th Gen Xeon, Platinum 8581C	2024
`n4d-standard-64`	AMD Turin, 5th Gen EPYC, 9B45	2025

Note: Processor labels combine Google Cloud platform names with checked-in `lscpu` metadata; release years identify replay diversity over Google Cloud generations.

TABLE III
ORIGINAL BENCHMARK VALIDITY RULES.

Benchmark	Original validity rule
GSO	Each selected generated test must still pass correctness/equivalence checks and keep a replay-time speedup of at least $1.2\times$ over the base commit. For final low-test tasks, tests with speedup above $1.1\times$ are treated as fallback-compatible, matching GSO's low-test construction fallback.
SWE-Perf	Each selected efficiency unit test must pass on the original and modified commits. After 20 timing repetitions and IQR outlier filtering, the recomputed Mann–Whitney minimum-gain value must remain above the dataset threshold: $\delta_i > 0.05$.
SWE-fficiency	The official workload and correctness guards must run successfully. The base mean workload runtime minus the expert-patched mean workload runtime must be larger than twice the post-edit workload-runtime standard deviation.

B. RQ1.1 Replay Evaluability

Results. We first check whether each reference patch replay completes and returns a usable score. Table IV counts tasks that complete in all three rounds on each replay machine. Most tasks are fully evaluable: GSO has 4/102 tasks that are not evaluable, SWE-Perf has 2/140, and SWE-fficiency has 0/498.

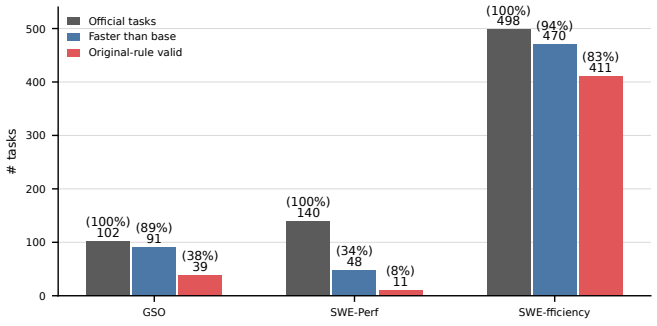

Fig. 1. Task counts after each replay check. Counts require passing all 12 machine-round replays (four machines across three rounds).

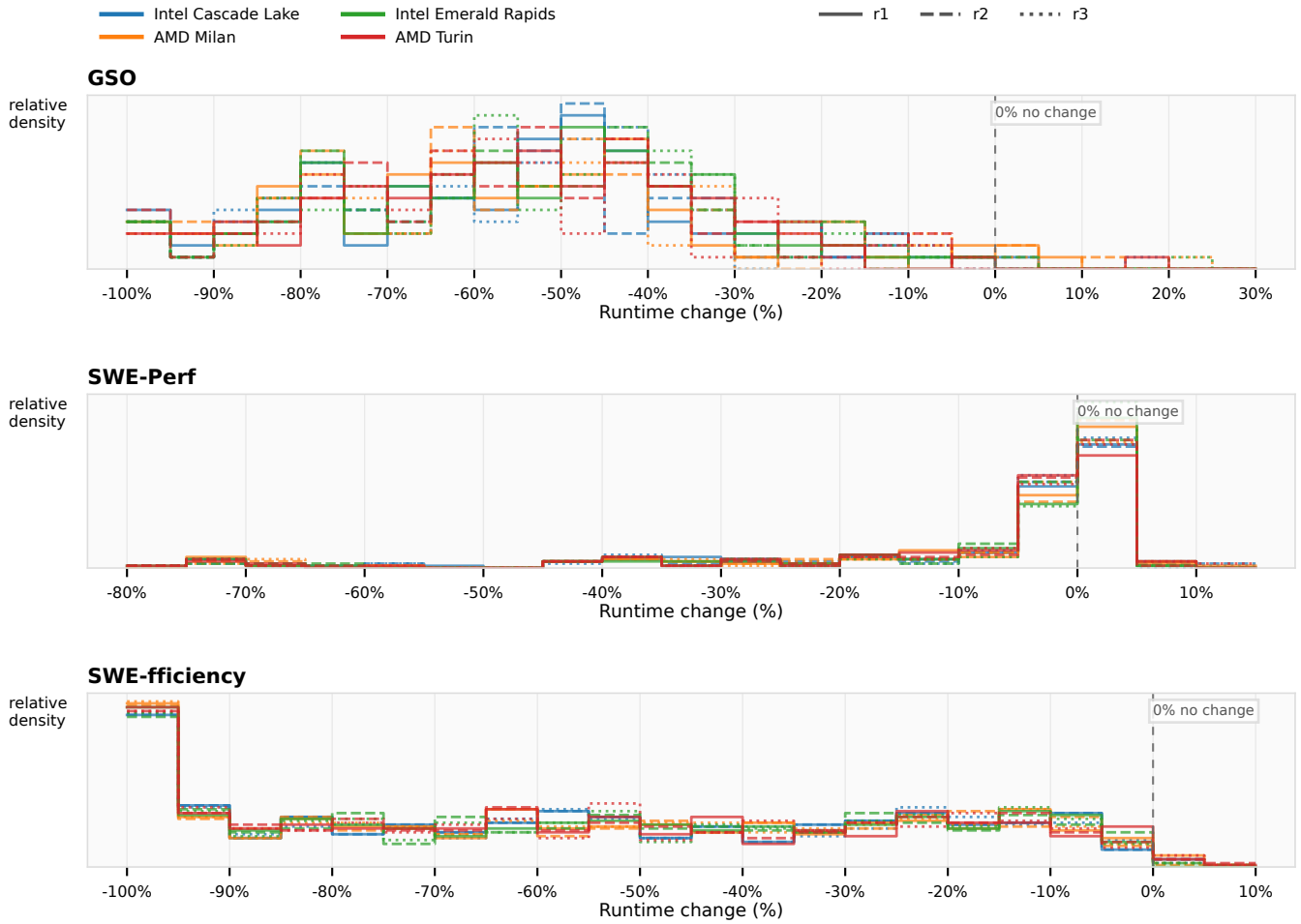

Fig. 2. Reference-patch runtime-change distributions by machine and replay round. Negative values mean the reference patch reduces runtime; positive values mean it increases runtime. SWE-Perf concentrates near the zero-change boundary, while GSO and SWE-fficiency show larger runtime reductions.

TABLE IV
REPLAY EVALUABILITY. COMPLETE-BY-MACHINE COUNTS REQUIRE ALL THREE ROUNDS ON N2/N2D/N4/N4D TO PRODUCE USABLE SCORES.

Benchmark	Official	Complete by machine	Not evaluable
GSO	102	99 / 98 / 99 / 99	4
SWE-Perf	140	138 / 138 / 138 / 138	2
SWE-fficiency	498	498 / 498 / 498 / 498	0

GSO failures come from one functional-equivalence failure, one missing external-data dependency, one image-decoding failure, and one x86_64 CPU-instruction portability failure. SWE-Perf failures come from missing dependencies in the evaluation image. SWE-fficiency has no replay-evaluability failure under our replay setup. We reported all four GSO cases and both SWE-Perf cases to the benchmark maintainers.

Finding RQ1.1

Most reference optimizations are replayable, but a small number fail because external resources, build/evaluation dependencies, or CPU-specific instructions do not port across machines.

C. RQ1.2 Reference Validity Under Replay

We next apply the two metrics from the experiment design. The *faster-than-base* metric only asks whether the reference patch beats the base commit in every machine-round replay. The *original-rule valid* metric is stricter: it reapplies the benchmark's own task-inclusion rule. Figure 1 summarizes both results using each benchmark's official task count as the denominator; not-evaluable tasks remain in that denominator and are counted as not passing the later timing checks.

Results. Figure 1 shows the deficiencies of reference patches. A simple runtime check shows that the reference patches in only 91/102 GSO tasks, 48/140 SWE-Perf tasks, and 470/498 SWE-fficiency tasks ran faster than the base code across machines. And, reapplying each benchmark's original validity rule leaves only 39, 11, and 411 valid tasks. The drop is the largest for SWE-Perf, where 129 reference patches no longer show statistically supported gain across machines.

TABLE V
RUNTIME SIGNAL AND REPLAY VARIATION.

Benchmark	Tasks	Median runtime change	Median runtime std.	Std./signal
GSO	98	-54.20%	3.81pp	0.07×
SWE-Perf	138	-0.03%	1.41pp	43.23×
SWE-fficiency	498	-56.04%	2.41pp	0.04×

Finding RQ1.2

All three benchmarks have reference patches that run slower than the base program in at least one cross-machine replay; applying each benchmark's own construction rule leaves even fewer tasks valid.

D. RQ1.3 Runtime Signal and Replay Variation

We next examine why tasks fail the original-rule check. Because the benchmarks report performance in different units, we convert every usable replay to a common *runtime change percentage*. Negative values mean the reference patch runs faster than the base program; positive values mean it runs slower. For GSO and SWE-fficiency, replay outputs are speedup ratios: a speedup of s means the reference patch runs s times faster than the base program ($s = T_{base}/T_{ref}$). We convert them to runtime change as $1/s-1$; for example, $1.20\times$ speedup becomes a -16.7% runtime change. For SWE-Perf, the benchmark already reports a significance-aware runtime-reduction score; we only flip its sign so that faster reference patches are negative on the shared runtime-change scale.

Table V reports both signal size and replay variation [14]–[16]. The median runtime change shows how far the reference patch is from no change, while the within-task standard deviation shows how much the 12 machine-round replays vary. We also report $\text{std.}/\text{signal} = \text{median std.}/|\text{median change}|$; larger values mean replay variation is large relative to the measured speed change.

Results. The table rules out a simple "more machine noise" explanation for SWE-Perf. Its median within-task standard deviation is 1.41 percentage points, lower than GSO and SWE-fficiency. The problem is signal size: the median SWE-Perf runtime change is only -0.03%, so even small timing shifts can cross zero or lose statistical support. Figure 2 shows the same pattern: 101/138 SWE-Perf tasks have median changes within five percentage points of zero. In contrast, GSO and SWE-fficiency have much larger median runtime reductions, so similar or larger absolute replay variation rarely changes the task-level conclusion.

Finding RQ1.3

SWE-Perf replay failures are mainly a small-signal problem: its reference patches cluster near no runtime change, while GSO and SWE-fficiency reference patches usually have larger speedup margins.

E. RQ1 Summary

The cross-machine replay gives a three-step answer. First, most reference patches are runnable across machines, so the main issue is not basic evaluability. Second, runnable does not imply benchmark-valid: all three benchmarks contain reference patches that become slower than the base program in at least one machine-round, and reapplying each benchmark's original construction rule narrows the valid set to 39 GSO tasks, 11 SWE-Perf tasks, and 411 SWE-fficiency tasks. Third, the failures are not explained simply by larger runtime noise. SWE-Perf reference patches cluster near zero runtime change, so small timing shifts can erase statistical support; GSO and SWE-fficiency usually retain much larger runtime reductions. One likely reason is benchmark construction: SWE-Perf derives efficiency checks from existing repository unit tests and accepts reference gains above a 5% threshold, whereas GSO and SWE-fficiency rely on more explicitly performance-oriented workloads or generated stress tests. Future performance-optimization benchmarks should validate that their workloads stress the optimized path strongly enough and that accepted reference patches have clear margins over runtime noise; otherwise, machine-level noise can be mistaken for a patch-level performance difference.

IV. RQ2: How Do Benchmark Scoring Rules Shape Submission Rankings?

This section audits how released task-level outputs become the scores in leaderboard rankings. The goal is to separate two effects: what submitted patches did on individual tasks, and how each benchmark scoring rule weights those outcomes into one leaderboard score.

A. Experiment Design

Benchmark selection. We include benchmarks that expose the artifacts needed for a metric audit: public submissions, per-task evaluation outcomes, and the final score used to rank submissions. As of our April 30, 2026 data-collection cutoff, GSO and SWE-fficiency released such public ranking data. SWE-Perf did not have comparable public agent-output data, so it is excluded from this analysis.

Scoring rules. To interpret rank shifts, the scoring rule must be stated at the task level: what does one task contribute to a submission's final score? The answer differs sharply across the two benchmarks.

GSO (binary reference-level gate). A submitted patch is counted as either a success or a failure [10]. For a submission m over N tasks, the submission score is

$$\text{OPT@1}(m) = 100 \cdot \frac{\#\text{reference-level successes}}{N}.$$

A reference-level success is a correct submitted patch whose speedup matches or exceeds the official reference patch.

- **Reward.** A correct reference-level patch adds one success, worth at most $100/N$ score points. For GSO's 102 tasks, one more success is worth 0.98 points.
- **Penalty.** An incorrect or below-reference patch adds zero successes; it loses $100/N$ points. Correct but below-reference speedups receive no partial credit.

SWE-fficiency (SpeedUp Ratio with harmonic mean). Instead of a binary gate, SWE-fficiency compares the submitted

patch's speedup with the reference patch's speedup [12]. Its task score is the SpeedUp Ratio (SR):

$$\mathrm{SR}_{m,i} = \frac{\mathrm{speedup}_{m,i}}{\mathrm{speedup}_{ref,i}}.$$

A value below 1 means the submitted patch is slower than the reference patch; a value above 1 means it beats the reference patch on that workload. The released task outputs treat failed or incorrect patches as no effective edit, which can place them far below the reference patch. The benchmark aggregates SR values with a harmonic mean [25], [26] after flooring each SR at 0.001:

$$\mathrm{HM}(m) = \frac{N}{\sum_i 1/\max(\mathrm{SR}_{m,i}, 0.001)}.$$

Here, each task contributes $1/t_i$ to the denominator, where $t_i = \max(\mathrm{SR}_{m,i}, 0.001)$. Lower denominator means higher final score.

- **Reward.** A task that matches the reference patch has $SR = 1$ and contributes one denominator unit. If it beats the reference patch with $SR = s > 1$, it contributes only $1/s$, so the reward is $1 - 1/s$ denominator units. For example, $SR = 2$ saves 0.5 units and $SR = 10$ saves 0.9 units; the reward is capped below one unit.
- **Penalty.** A below-reference task exceeds the $SR = 1$ baseline by $1/SR - 1$ denominator units: $SR = 0.5$ adds 1, $SR = 0.01$ adds 99, and the official floor $f = 0.001$ adds 999. In the 498-task SWE-fficiency evaluation, one floor-level task can outweigh a full reference-matching denominator.

B. RQ2.1 Cross-Benchmark Metric Sensitivity

We first ask whether the shared public submissions rank similarly across the two benchmarks, and whether the strong/weak judgments change when the same task outputs are scored by another benchmark's rule. The comparison uses the eight public submissions that appear on both GSO and SWE-fficiency. For each benchmark's released per-task outputs, we keep the outputs fixed and change only the aggregation rule: SWE-fficiency outputs are rescored with the GSO reference-level gate, and available GSO outputs are rescored with SWE-fficiency's harmonic rule.

Results. The official ranks are not stable across benchmarks. The eight shared submissions form $\binom{8}{2} = 28$ unique head-to-head pairs when each pair is compared once; the two official leaderboards disagree on 9 of them, and the GPT-5-labeled submission moves by five positions (Table VI).

Changing only the scoring rule explains part, but not all, of the mismatch. Re-aggregating the same SWE-fficiency task outputs with a GSO-style reference-level pass rate raises Spearman rank correlation with GSO from 0.452 to 0.762 and reduces discordant pairs from 9 to 6. In the other direction, applying SWE-fficiency's harmonic scoring to available GSO per-task submission reports gives a Spearman rank correlation of 0.238 against the official SWE-fficiency ranking and flips 11/28 head-to-head pairs. This second rescoring makes agreement weaker than the official comparison and flips more pairs, which suggests that the harmonic scoring rule itself is a sensitive part of the benchmark. We therefore treat these rescoring results as diagnostics rather than replacement rankings. They show that scoring rules affect the ordering, while the task set and per-task outputs also matter (Table VII).

TABLE VI
OFFICIAL RANKS FOR SHARED SUBMISSIONS.

Model label	GSO		SWE-fficiency		Move
	Rank	Score	Rank	Score	
Claude Opus 4.6	1	41.18	3	0.1553	↓2
GPT-5.2	2	27.45	4	0.1482	↓2
Claude Opus 4.5	3	26.47	1	0.2250	↑2
Gemini 3 Pro	4	18.63	7	0.1024	↓3
Claude Sonnet 4.5	5	14.71	5	0.1162	–
Gemini 3 Flash	6	9.80	6	0.1056	–
GPT-5	7	6.86	2	0.1571	↑5
Gemini 2.5 Pro	8	3.92	8	0.0313	–

Note. Move is GSO rank → SWE-fficiency rank.

TABLE VII
SCORING-RULE DIAGNOSTICS FOR THE EIGHT SHARED SUBMISSIONS.

Comparison	Spearman corr.	Pair flips	Rank movement
Official GSO vs. official SWE-fficiency	0.452	9/28	max 5
Both with GSO scoring	0.762	6/28	max 3
Both with SWE-fficiency scoring	0.238	11/28	max 5

Note. Spearman corr. compares the two rank orderings (-1 reverse, 1 identical) [27], [28]; pair flips count disagreements among the 28 unique head-to-head pairs.

Finding RQ2.1

The same submissions can rank differently across performance benchmarks because scoring rules decide which task outcomes count and how strongly they shape the final order.

C. RQ2.2 Low-Speedup Tail Dominance

The scoring-rule comparison above suggests that SWE-fficiency's harmonic scoring rule deserves a closer audit. The concern is not simply that low-SR tasks receive low scores. In a harmonic mean, each task enters the denominator as $1/\max(SR, 0.001)$, so a very low-SR task can have much more leverage than a typical task. To measure this leverage directly, we compute each task's share of the score denominator,

$$w_i = \frac{1/\max(SR_i, 0.001)}{\sum_j 1/\max(SR_j, 0.001)},$$

then sort tasks by w_i within each shared submission and ask how much score weight is carried by the worst 1, 5, and 10 tasks.

Results. Figure 3 shows that Worst-1 tasks carry 6.3–33.6%, worst-5 tasks carry 31.4–73.1%, and worst-10 tasks carry 58.5–82.8% of the official score denominator. This is large for a 498-task benchmark: a small set of low-SR tasks can explain most of the aggregate score weight. For example, one near-floor task in Claude Opus 4.5 has raw SR 0.00134 and carries 33.6% of the submission's score weight.

This does not mean severe slowdowns should be ignored. Penalizing them is a reasonable benchmark choice. The issue

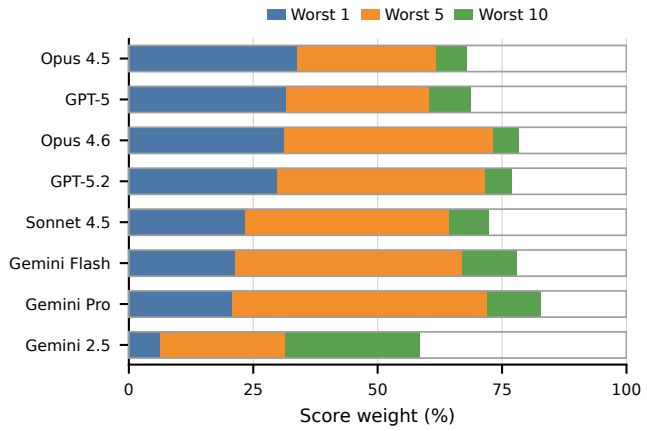

Fig. 3. Score weight of the worst 1, 5, and 10 tasks under SWE-fficiency's harmonic mean.

is interpretability: when 10 tasks carry more than half of the denominator, the official harmonic mean behaves partly as a metric driven by the worst tasks. A rank difference may therefore reflect broad performance across the benchmark, or it may mostly reflect a few near-floor failures. This distinction matters when interpreting the ranking.

Finding RQ2.2

SWE-fficiency's final score is dominated by a few very low-speedup tasks: the worst ten tasks carry 58.5–82.8% of score weight, and one near-floor task can carry one third of a submission's score weight.

D. RQ2.3 Exploring Reasonable Penalty Bounds

After measuring tail dominance, we ask how large the penalty for one slow or invalid task should be. The official SWE-fficiency floor $f = 0.001$ allows one near-zero SpeedUp Ratio (SR) task to contribute 1000 denominator units, or 999 more than an $SR = 1$ reference-matching task. In a 498-task benchmark, this means one severe failure can dominate many ordinary tasks, making the leaderboard score hard to interpret. We therefore explore a simple bounded-penalty diagnostic: keep harmonic aggregation, but make the largest below-reference penalty comparable to the largest possible above-reference reward. Because the harmonic reward is capped below one denominator unit, we cap the extra penalty at one unit as well. This corresponds to raising the floor to 0.5: a very low-SR task contributes at most two denominator units, only one more than a neutral task. This probes how rankings change when slow tasks are penalized but cannot dominate.

Results. Table VIII shows that this design choice changes six of eight submission ranks and flips 8/28 pairwise submission orders relative to the official SWE-fficiency ranking. With the cap, ranks become closer to median SR and above-reference task counts: Claude Opus 4.6 and GPT-5.2 move above Claude Opus 4.5, while GPT-5 drops. The point is not that $f = 0.5$ is the right constant. It is that the leaderboard order depends on the penalty budget given to one bad task, which readers should keep in mind when interpreting the score.

TABLE VIII
BOUNDED-PENALTY DIAGNOSTIC WITH FLOOR $f = 0.5$.

Rank	Model	Move	Official HM	HM ($f = 0.5$)	Med. SR	# SR>1
1	Claude Opus 4.6	↑2	0.1553	0.952	1.006	258
2	GPT-5.2	↑2	0.1482	0.944	1.004	259
3	Claude Opus 4.5	↓2	0.2250	0.904	0.986	212
4	Gemini 3 Pro	↑3	0.1024	0.888	0.993	235
5	GPT-5	↓3	0.1571	0.867	0.950	228
6	Gemini 3 Flash	–	0.1056	0.818	0.919	185
7	Claude Sonnet 4.5	↓2	0.1162	0.767	0.858	164
8	Gemini 2.5 Pro	–	0.0313	0.613	0.497	38

Note. Move is relative to the official SWE-fficiency rank (floor $f = 0.001$).

Finding RQ2.3

Changing the single-task penalty cap reshuffles the SWE-fficiency leaderboard: 6/8 ranks move and 8/28 pairwise orders flip. Leaderboard scores reflect both submission performance and penalty design.

E. RQ2 Summary

Taken together, the ranking results do not directly measure agent capability. They combine what submissions did on individual tasks with how the benchmark turns those task outcomes into one leaderboard score. Among the eight shared submissions, the official GSO and SWE-fficiency rankings disagree on 9 of the 28 pairwise orders. When the same task outputs are re-scored with another rule, ranks still move, showing that both task outcomes and scoring design shape the final order. The clearest difference is how much one bad task can hurt a submission. GSO gives each task one equal success/failure vote. SWE-fficiency uses continuous speedup ratios, but its harmonic mean with a 0.001 floor can let a few near-zero tasks dominate the final score: the worst ten tasks carry 58.5–82.8% of score weight. When we cap the maximum damage from one task, 6/8 ranks move and 8/28 pairwise comparisons flip. Thus, SWE-fficiency is best read as a strict, tail-sensitive score: its leaderboard reflects both submission behavior and the metric's penalty design.

V. RQ3: Do Benchmark Tasks Remain Hard for Top Submissions?

In practice, current agent systems often use multi-agent workflows rather than relying on a single run [24], [29]–[31]. However, the public leaderboards we analyze rank individual OpenHands-based submissions, each pairing the agent scaffold with one underlying model [2]. This mismatch matters because different submissions may solve different tasks. We therefore look across 10 public submissions for each replay-valid task and ask whether any submission reaches the reference patch. This gives an optimistic estimate of how many tasks are covered by at least one strong public submission; it does not mean that one submission reaches the reference on all of them.

A. RQ3.1 How Many Tasks Remain Below Reference?

The first step checks the replay-valid task subset: 39 GSO tasks and 411 SWE-fficiency tasks. SWE-Perf is excluded

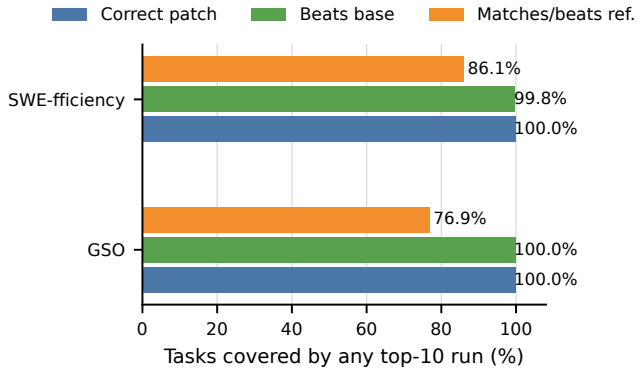

Fig. 4. Task outcomes across 10 public submissions on the replay-valid subset (39 GSO, 411 SWE-fficiency tasks).

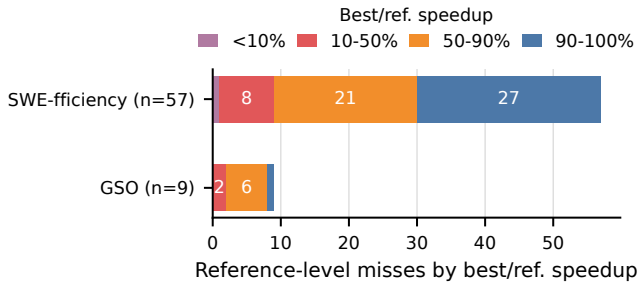

Fig. 5. Best public patch speed relative to the reference patch for the 66 tasks that do not reach reference-patch speed.

because it has only 11 replay-valid tasks and no comparable released public agent-solution data. For each task, we look across 10 public submissions and record whether at least one submission reaches three milestones: passing tests, beating the unmodified base program, and matching or beating the reference patch.

Results. Figure 4 shows that test passing and base-program speedup are nearly complete when we look across 10 public submissions. Every replay-valid GSO and SWE-fficiency task has at least one passing public patch, and 449/450 tasks have a passing patch that beats the base program. The remaining signal appears only at the stricter reference-level gate: at least one of the 10 public submissions matches the reference on 384/450 tasks, leaving 66 tasks not reaching reference speed, 9/39 for GSO and 57/411 for SWE-fficiency.

Finding RQ3.1

Across 10 public submissions, most replay-valid tasks already have a strong public solution: 384/450 match or beat the reference patch, all 450 have a passing patch, and only 66 remain below reference.

B. RQ3.2 Are the Remaining Tasks Truly Unsolved?

The 66 tasks that still do not reach the reference-patch speed could be genuinely unsolved, or they could already have useful public patches that are simply not as fast as the reference patch. For each task, we inspect the *best public patch*: among the 10 public submissions, the patch with the largest replayed speedup relative to the reference patch. Every task has a correct public patch, 65/66 already run faster than the base program, and only one task has a best patch that passes tests without improving runtime. Individual attempts still fail, but this task-level selection removes most task-level blockers: across the 660 attempts behind these 66 tasks, 160 fail functionality or validation, 3 do not complete, and 46 pass correctness without improving runtime.

Results. Figure 5 shows that most remaining tasks are close to the reference patch rather than broken outright. For GSO, the best public patch reaches a median 85.3% of the reference patch's speedup and would need 1.17× more speedup to match it. For SWE-fficiency, the median is 87.9%, leaving 1.14×. Many tasks are near the reference: 27 SWE-fficiency tasks reach 90–100% of the reference patch's speedup. The tail still matters, however: 8 SWE-fficiency tasks reach only 10–50% and one is below 10%.

Finding RQ3.2

Remaining tasks are rarely complete failures: all have a correct public patch, 65/66 beat the base program, and the median best patch reaches 85.3%/87.9% of reference speedup on the two benchmarks.

C. RQ3.3 Does the Hard Tail Require the Reference Strategy?

The previous results show that the remaining tasks are often not basic correctness failures: the best public patch usually passes tests and improves runtime, but still trails the reference patch. A natural hypothesis is that strategy choice explains this gap. If high performance depends on targeting the right hot operation or using the right optimization mechanism, public patches that use a different strategy from the reference patch should leave larger speedup gaps. This subsection explores that hypothesis by comparing each best public patch with its reference patch. The reference is a diagnostic baseline, not an imitation target: a different strategy is acceptable if it reaches similar speedup.

Annotation method. For each of the 66 tasks that still do not reach reference speed, we compare the reference patch with the best public patch defined above. We use the high-level optimization categories from Peng et al.'s taxonomy [32], [33]. We reuse their automated annotation script with GPT-5.5 as the annotation model to assign one category to each reference patch and each best public patch. We schema-check the labels, join them into reference-vs.-public pairs, and mark whether the best public patch uses the same category, a different category, or no visible production optimization. We report both category alignment and the remaining gap, measured as best-public speedup divided by reference-patch speedup. We manually sanity-check the labels against patch diffs.

Results. Table IX shows both the shape of the remaining tasks and how often the best public patch matches the reference category. Algorithm reference patches are the largest group and are matched most often (21/31). Structure is split (7/16 matched), while memory-heavy reference patches are

TABLE IX
REFERENCE-CATEGORY ALIGNMENT.

Reference category	Ref. pairs	Same cat.	Diff. cat.
Algorithm	31	21/31	10/31
Structure	16	7/16	9/16
Memory	12	3/12	9/12
Build	3	0/3	3/3
Control	3	1/3	2/3
Data struct.	1	0/1	1/1
Total	66	32/66	34/66

usually approached through another category (9/12 different). Overall, 32/66 best public patches use the same high-level category as the reference patch, and 34/66 use a different category or show no visible production optimization.

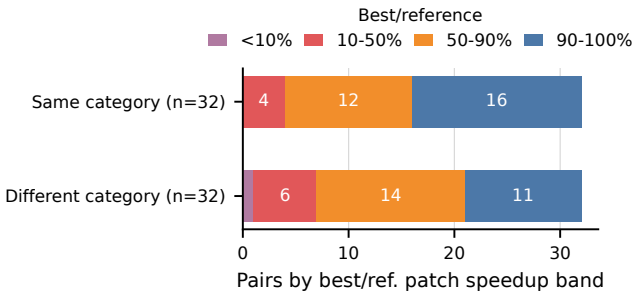

Fig. 6. Best public/reference speedup ranges by high-level category alignment.

Figures 6 and 7 compare the remaining speed gap by category alignment. For this speedup comparison, we exclude the two cases without visible production optimization, leaving 32 same-category and 32 different-category public optimizations. Same-category patches are closer to the reference on median (89.8% vs. 81.1%; remaining multiplier 1.12× vs. 1.23×), but the two groups overlap substantially: 16/32 same-category cases still remain below 90% of the reference speedup, while 11/32 different-category cases reach 90–100%. Same-category cases even include four below-50% gaps. Category alignment is therefore only a partial signal, not a sufficient condition for near-reference performance. The remaining gap often reflects implementation depth: agents often find a useful optimization idea but do not apply it broadly or carefully enough to match the reference patch.

Finding RQ3.3

Optimization strategy mismatch is not the main reason tasks stay below reference: among 66 tasks, 32 same-category public patches remain slower, while 11 different-category patches reach 90–100% of reference speed.

D. RQ3 Summary

Looking across 10 public submissions shows that many replay-valid GSO and SWE-fficiency tasks are already solved by at least one public submission. Across 450 tasks, 384 already match or beat the reference patch, all 450 have a passing public patch, and 449 have a patch that beats the base program. The remaining 66 tasks are therefore not mostly basic correctness failures. They usually already have a correct and runtime-improving public patch, with median best-patch speed reaching 85.3% of the reference speedup on GSO and 87.9% on SWE-fficiency.

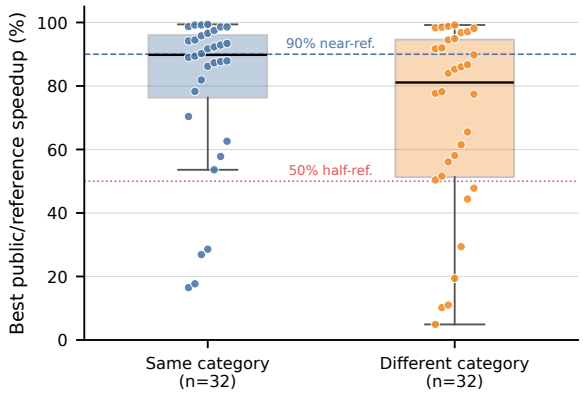

Fig. 7. Best public/reference speedup by high-level strategy alignment.

The remaining speed gap also does not appear to be mainly a broad strategy-selection problem. In 32/66 tasks, the best public patch uses the same high-level optimization category as the reference patch; in 11 more, a different category still reaches 90–100% of the reference speedup. This suggests that public submissions often identify a useful performance direction, not merely a passing edit. What remains is deeper optimization after the first correct speedup: without seeing the reference patch, the runs may stop short of the extra iteration needed to reach reference-patch speed.

VI. DISCUSSION

Our results suggest three practical implications for reading and designing performance-optimization benchmarks.

For benchmark users: separate signal from measurement design. Leaderboard scores are useful, but our results show that current benchmark scores should not be read as direct labels of agent capability. RQ1 shows that reference-patch evidence can change across machines, especially when the runtime gain is close to zero. This means a reported optimization can be hard to distinguish from runtime noise. RQ2 shows a second layer of interpretation: even when task outputs are fixed, the aggregation rule can decide how much one bad task matters. Under SWE-fficiency's leaderboard scoring rule, a few low-speedup tasks can account for most of the score weight, so a rank difference may reflect broad task performance, a small set of severe failures, or both. Users of these benchmarks should therefore look beyond the leaderboard rank. At minimum, reports should distinguish cross-machine verified tasks from unstable tasks, show per-task score weight, and compare submissions under aggregation rules that match the use case. For some users, strong worst-case penalties are appropriate; for others, median-speedup or reference-level coverage is more informative.

For agent builders: interpret multi-agent gains at the task level. A single submitted run remains a valid and strict competition setting, but it is not the same as asking how many tasks today's public submission outputs still leave unsolved. RQ3 shows that, on the replay-valid GSO and SWE-fficiency tasks, at least one of the 10 public submissions already reaches the reference patch on 384/450 tasks, and almost every remaining task has a correct patch that is faster than the base program. This matters for multi-agent workflows that can draw on several agent configurations [34], [35]. If a new agent configuration is evaluated only by making more tasks reach the reference patch, the remaining room is narrow and the measured improvement may depend heavily on a small set of tasks. This does not make the benchmarks useless; it changes what they are good for. They are still useful for studying the last step from "faster than base" to "as fast as the reference patch," but less suitable as the only evidence of general performance-engineering ability.

For future benchmark designers: move closer to the full performance engineering problem. Current benchmarks provide executable workloads and reproducible task artifacts, but they often simplify the task by exposing the optimized code region, workload, or stress test to the agent. Production performance work usually starts earlier: engineers inspect profiles, flame graphs, traces, latency breakdowns, or dashboards, then choose which hot operation in a large codebase is worth changing, infer which measurement matters, and judge whether a local edit will help the real workload without a ready-made stress test for immediate feedback. Future benchmarks should preserve executable replay while adding this diagnostic layer: hotspot localization from profiles or traces, ambiguous optimization targets, validation against workloads not fully visible during patch search, and resource metrics beyond one runtime number. CPU time, latency, allocation behavior, memory footprint, and regression risk all matter, and optimizing one can hurt another [36]. The goal is not to discard today's benchmarks, but to make the next ones test a harder workflow: finding the bottleneck, choosing a justified optimization, implementing it across the right code paths, and proving that it improves the workload that matters.

VII. Threats to Validity

Internal validity. Our replays and metric recomputations may still be affected by implementation choices, hardware variation, and released artifact quality. The replay analysis therefore uses a strict all-replay rule: a task is replay-valid only when the reference patch satisfies the benchmark's original construction rule in all 12 machine-round replays. This conservative rule may undercount usable tasks, but it protects later analyses from unstable reference signals.

External validity. The results cover three recent repository-level performance-optimization benchmarks and specific public leaderboard snapshots, not all coding-agent or performance-engineering settings. Our analysis across 10 public submissions is also a task-coverage proxy, not a newly engineered multi-agent workflow or a claim about any single submitted configuration. Future underlying models, agent scaffolds, hardware, and benchmark policies may change the exact counts.

Data construction validity. We rely on each benchmark's released tasks, reference patches, public submissions, and scoring records. SWE-Perf is excluded from the metric and RQ3 task analyses because comparable public agent-output artifacts were unavailable.

VIII. Related Work

Code-efficiency benchmarks have expanded from small code units to full repositories. PIE studies performance-improving edits, while EffiBench, Mercury, ENAMEL, ECCO, EvalPerf/DPE, and ACECode evaluate whether generated or revised code remains correct while improving efficiency [4]–[6], [37]–[41]. COFFE adds function- and file-level tracks, while EffiBench-X, KernelBench, AlgoTune, and PerfCodeBench broaden evaluation across languages, GPU kernels, numerical programs, and system-level high-performance code [7], [42]–[45]. At the repository and project level, GSO, SWE-Perf, and SWE-fficiency evaluate edits to real repositories with executable workloads, reference patches, and benchmark-specific metrics [10]–[12]. PerfBench, PEACE/PeacExec, ISO-Bench, FormulaCode, and CppPerf respectively cover real-world performance bugs, project-level efficiency optimization with hybrid editing, inference workloads, large-codebase agent optimization, and performance-improving C++ commits [46]–[50]. Together, these benchmarks show a clear shift from function-level efficiency to repository-scale performance engineering [51]–[53].

In contrast, we do not introduce another benchmark or report a new agent leaderboard. We audit the released artifacts behind existing repository-level benchmarks: whether official reference patches replay reliably, how scoring rules change public-submission rankings, and which replay-valid tasks still separate public submissions from reference patches.

IX. Conclusion

Performance-optimization benchmarks now influence how coding agents are judged, but their scores are not self-explanatory. Our audit of three recent repository-level benchmarks shows why. After replaying 740 official reference patches across four cloud machines and three rounds, only 39/102 GSO tasks, 11/140 SWE-Perf tasks, and 411/498 SWE-fficiency tasks kept their original validity signal in every replay. Leaderboard ranks also depend on the scoring rule: among the eight public submissions shared by GSO and SWE-fficiency, the official rankings disagree on 9/28 pairwise orders, and the worst ten low-speedup SWE-fficiency tasks carry 58.5%–82.8% of the score weight. Finally, many replay-valid tasks are already covered by at least one public submission: 384/450 match or beat the reference patch, and 449/450 beat the base program. For the remaining tasks, the problem is usually not finding any working optimization, but closing the last speedup gap to the reference patch.

The next generation of performance-optimization benchmarks should therefore test more than patching a pre-specified

entry point under a fixed workload. They should ask agents to reason from performance symptoms or profiles, choose optimization targets, and validate runtime improvements together with memory, latency, and other resource costs.

REFERENCES

[1] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 50 528–50 652.

[2] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "OpenHands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations*, 2025.

[3] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '24. ACM, September 2024, pp. 1592–1604. [Online]. Available: http://dx.doi.org/10.1145/3650212.3680384

[4] A. G. Shypula, A. Madaan, Y. Zeng, U. Alon, J. R. Gardner, Y. Yang, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," in *The Twelfth International Conference on Learning Representations*, 2024.

[5] D. Huang, Y. Qing, W. Shang, H. Cui, and J. Zhang, "EffiBench: Benchmarking the efficiency of automatically generated code," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 11 506–11 544.

[6] M. Du, L. A. Tuan, B. Ji, Q. Liu, and S.-K. Ng, "Mercury: A code efficiency benchmark for code large language models," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 16 601–16 622.

[7] Y. Peng, J. Wan, Y. Li, and X. Ren, "COFFE: A code efficiency benchmark for code generation," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 242–265, 2025.

[8] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[9] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158

[10] M. Shetty, N. Jain, J. Liu, V. Kethanaboyina, K. Sen, and I. Stoica, "GSO: Challenging software optimization tasks for evaluating SWE-agents," in *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, Eds., vol. 38. Curran Associates, Inc., 2025.

[11] X. He, Q. Liu, M. Du, L. Yan, Z. Fan, Y. Huang, Z. Yuan, and Z. Ma, "Swe-perf: Can language models optimize code performance on real-world repositories?" in *Forty-third International Conference on Machine Learning*, 2026. [Online]. Available: https://openreview.net/forum?id=pNxw4i5Dp2

[12] J. J. Ma, M. Hashemi, A. Yazdanbakhsh, K. Swersky, O. Press, E. Li, V. J. Reddi, and P. Ranganathan, "Swe-fficiency: Can language models optimize real-world repositories on real workloads?" in *Forty-third International Conference on Machine Learning*, 2026. [Online]. Available: https://openreview.net/forum?id=J1lgJyP4Tm

[13] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[14] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, ser. OOPSLA07. ACM, Oct. 2007, pp. 57–76. [Online]. Available: http://dx.doi.org/10.1145/1297027.1297033

[15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS09. ACM, Mar. 2009, pp. 265–276. [Online]. Available: http://dx.doi.org/10.1145/1508244.1508275

[16] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 international symposium on memory management*, ser. ISMM '13. ACM, June 2013, pp. 63–74.

[17] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *ACM Transactions on Software Engineering and Methodology*, Mar. 2026. [Online]. Available: http://dx.doi.org/10.1145/3796507

[18] Y. Wang, W. Zhong, Y. Huang, E. Shi, M. Yang, J. Chen, H. Li, Y. Ma, Q. Wang, and Z. Zheng, "Agents in software engineering: survey, landscape, and vision," *Automated Software Engineering*, vol. 32, no. 2, Aug. 2025. [Online]. Available: http://dx.doi.org/10.1007/s10515-025-00544-2

[19] Z. Chen and L. Jiang, "Evaluating software development agents: Patch patterns, code quality, and issue complexity in real-world github scenarios," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2025, pp. 657–668. [Online]. Available: http://dx.doi.org/10.1109/SANER64311.2025.00068

[20] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[21] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=iaYcJKpY2B_

[22] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[23] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 68 539–68 551. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf

[24] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=BAakY1hNKS

[25] P. S. Bullen, *Handbook of Means and Their Inequalities*. Springer Netherlands, 2003. [Online]. Available: http://dx.doi.org/10.1007/978-94-017-0399-4

[26] P. De, "The arithmetic mean - geometric mean - harmonic mean: Inequalities and a spectrum of applications," *Resonance*, vol. 21, no. 12, pp. 1119–1133, Dec. 2016. [Online]. Available: http://dx.doi.org/10.1007/s12045-016-0423-4

[27] C. Spearman, "The proof and measurement of association between two

things," *The American Journal of Psychology*, vol. 15, no. 1, p. 72, Jan. 1904. [Online]. Available: http://dx.doi.org/10.2307/1412159

[28] M. G. KENDALL, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1-2, pp. 81–93, Jun. 1938. [Online]. Available: http://dx.doi.org/10.1093/biomet/30.1-2.81

[29] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X

[30] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 11 809–11 822. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf

[31] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 8634–8652. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf

[32] H. Peng, A. Gupte, R. Hasler, N. J. Eliopoulos, C.-C. Ho, R. Mantri, L. Deng, K. Läufer, G. K. Thiruvathukal, and J. C. Davis, "Sysllmatic: Large language models are software system optimizers," *Journal of Systems and Software*, vol. 240, p. 112929, Oct. 2026. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2026.112929

[33] H. Peng, A. Zhong, R. A. C. Méndez, K. G. Kalu, and J. C. Davis, "How do agents perform code optimization? an empirical study," 2025, accepted to MSR '26: Proceedings of the 23rd International Conference on Mining Software Repositories. [Online]. Available: https://arxiv.org/abs/2512.21757

[34] W. Li, Z. Chen, J. Lin, H. Cao, W. Han, S. Liang, Z. Zhang, K. Dong, D. Li, C. Zhang, and Y. Liu, "Reinforcement learning foundations for deep research systems: A survey," 2025. [Online]. Available: https://arxiv.org/abs/2509.06733

[35] W. Ma, Y. Li, Z. Chen, Y. Liu, L. Jiang, Q. Hu, and J. Tao, *AgentGuard: An Active Threat Discovery System for Package Confusion Using Multi-Agent Collaboration*. Springer Nature Singapore, 2026, pp. 69–83. [Online]. Available: http://dx.doi.org/10.1007/978-981-95-7820-7_5

[36] Z. Sun, Z. Lin, Z. Chen, C. Yang, M. Zhou, L. Li, and D. Lo, "Executing as you generate: Hiding execution latency in llm code interpreters," 2026. [Online]. Available: https://arxiv.org/abs/2604.00491

[37] R. Qiu, W. W. Zeng, J. Ezick, C. Lott, and H. Tong, "How efficient is LLM-generated code? a rigorous & high-standard benchmark," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=suz4utPr9Y

[38] S. Waghjale, V. Veerendranath, Z. Wang, and D. 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*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 15 362–15 376. [Online]. Available: https://aclanthology.org/2024.emnlp-main.859/

[39] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=IBCBMeAhmC

[40] C. Yang, H. J. Kang, J. Shi, and D. Lo, "ACECode: A reinforcement learning framework for aligning code efficiency and correctness in code language models," 2024. [Online]. Available: https://arxiv.org/abs/2412.17264

[41] M. Harman, J. Ritchey, I. Harper, S. Sengupta, K. Mao, A. Gulati, C. Foster, and H. Robert, "Mutation-guided llm-based test generation at meta," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. ACM, June 2025, pp. 180–191.

[42] Y. Qing, B. Zhu, M. Du, Z. Guo, T. Y. Zhuo, Q. Zhang, J. M. Zhang, H. Cui, S.-M. Yiu, D. Huang, S.-K. Ng, and A. T. Luu, "EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code," in *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, Eds., vol. 38. Curran Associates, Inc., 2025.

[43] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Re, and A. Mirhoseini, "KernelBench: Can LLMs write efficient GPU kernels?" in *Proceedings of the 42nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu, Eds., vol. 267. PMLR, 13–19 Jul 2025, pp. 47 356–47 415.

[44] O. Press, B. Amos, H. Zhao, Y. Wu, S. K. Ainsworth, D. Krupke, P. Kidger, T. Sajed, B. Stellato, J. Park, N. Bosch, E. Meril, A. Steppi, A. Zharmagambetov, F. Zhang, D. Pérez-Piñeiro, A. Mercurio, N. Zhan, T. Abramovich, K. Lieret, H. Zhang, S. Huang, M. Bethge, and O. Press, "AlgoTune: Can language models speed up general-purpose numerical programs?" in *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, Eds., vol. 38. Curran Associates, Inc., 2025.

[45] H. Jing, W. Hu, H. Shi, H. Yang, S. Zhang, S. Chen, H. Li, and Y. Song, "PerfCodeBench: Benchmarking LLMs for system-level high-performance code optimization," 2026. [Online]. Available: https://arxiv.org/abs/2605.15222

[46] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "PerfBench: Can agents resolve real-world performance bugs?" in *Proceedings of the 2026 International Workshop on Agentic Engineering*, ser. AGENT '26. ACM, Apr. 2026, pp. 165–172.

[47] X. Ren, J. Wan, Y. Peng, Z. Liu, M. Liang, D. Chen, W. Jiang, and Y. Li, "PEACE: Towards efficient project-level efficiency optimization via hybrid code editing," in *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2025, pp. 1831–1843.

[48] A. Nangia, S. Mishra, A. Gokrani, and P. Chopra, "ISO-bench: Can coding agents optimize real-world inference workloads?" in *VerifAI-2: The Second Workshop on AI Verification in the Wild*, 2026. [Online]. Available: https://openreview.net/forum?id=gPfqEnaKCy

[49] A. Sehgal, J. Hou, A. Sarkar, I. Mantripragada, S. Chaudhuri, J. J. Sun, and Y. Yue, "FormulaCode: Evaluating agentic optimization on large codebases," in *Forty-third International Conference on Machine Learning*, 2026. [Online]. Available: https://openreview.net/forum?id=WArbqRUsAe

[50] T. Ho, K. Etemadi, and Z. Su, "CppPerf: An automated pipeline and dataset for performance-improving C++ commits," 2026. [Online]. Available: https://arxiv.org/abs/2605.10890

[51] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, Nov. 2024. [Online]. Available: http://dx.doi.org/10.1145/3695988

[52] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, May 2023, pp. 31–53. [Online]. Available: http://dx.doi.org/10.1109/ICSE-FoSE59343.2023.00008

[53] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, S. Yu, W. Sun, Y. Yang, and Z. Chen, "A survey on large language models for software engineering," *Science China Information Sciences*, vol. 69, no. 4, Mar. 2026. [Online]. Available: http://dx.doi.org/10.1007/s11432-025-4670-0