SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories?

Xinyi He[1*] **Qian Liu**[2†] **Mingzhe Du**[3] **Lin Yan**[2] **Zhijie Fan**[2] **Yiming Huang**[4*]
Zejian Yuan[1] **Zejun Ma**[2]
[1] Xi'an Jiaotong University, [2] TikTok,
[3] National University of Singapore, [4] University of California San Diego

Abstract

Code performance optimization is paramount in real-world software engineering and critical for production-level systems. While Large Language Models (LLMs) have demonstrated impressive capabilities in code generation and bug fixing, their proficiency in enhancing code performance at the repository level remains largely unexplored. To address this gap, we introduce SWE-Perf, the first benchmark specifically designed to systematically evaluate LLMs on code performance optimization tasks within authentic repository contexts. SWE-Perf comprises 140 carefully curated instances, each derived from performance-improving pull requests from popular GitHub repositories. Each benchmark instance includes the relevant codebase, target functions, performance-related tests, expert-authored patches, and executable environments. Through a comprehensive evaluation of representative methods that span file-level and repo-level approaches (e.g., Agentless and OpenHands), we reveal a substantial capability gap between existing LLMs and expert-level optimization performance, highlighting critical research opportunities in this emerging field.

1 Introduction

Recent advances in Large Language Models (LLMs) have significantly enhanced automated code generation and software development assistance, exemplified by tools like GitHub Copilot [10] and Cursor [2]. This progress has spurred growing interest in repository-level software engineering challenges in real-world settings [7]. Recent work has introduced multiple benchmarks for evaluating LLM code correctness, including SWE-Bench [7] and SWE-Dev [4], as well as frameworks such as AgentLess [20] and OpenHands [19] that aim to improve the accuracy of LLM-generated code. However, while correctness is foundational in production environments, performance optimization often yields more profound system-wide benefits [14, 9, 17]. Performance optimization is a task traditionally requiring specialized human expertise and poses significant challenges for LLMs. This raises a critical research question: **Can language models effectively optimize code performance in real-world repositories?**

While software engineering and code performance optimization have progressed significantly as distinct fields, current benchmarks struggle to evaluate tasks demanding their integration, especially for repository-level code performance optimization. Repository-level software engineering benchmarks [5, 7, 13, 4, 12] face particular challenges in supporting open-source code performance optimization. Evaluating whether LLMs can achieve meaningful optimizations is hindered by the lack of human reference implementations for "optimal" efficiency, making it unclear whether code

[*] This work was conducted during Xinyi and Yiming's internship at TikTok.
[†] Corresponding author.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

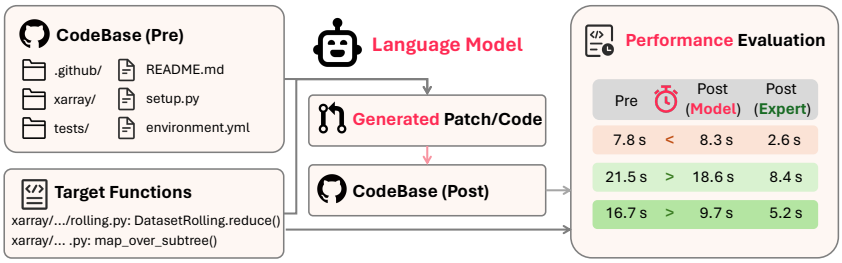

Figure 1: The workflow of our SWE-Perf benchmark which evaluates code performance optimization capabilities of language models. The benchmark evaluates language models by providing source code and performance-related tests, challenging them to generate optimized patches. Model performance is evaluated by the runtime gains on the tests, with expert performance as reference.

can be improved further. Meanwhile, existing benchmarks focused on code performance [3, 6, 8] primarily target function-level optimizations for algorithmic problems, consequently neglecting the complexities of repository-scale optimization. This represents a critical departure from real-world practice, where collaborative improvements between files and modules typically unlock far greater optimization potential than isolated function-level changes.

To address these gaps, we propose **SWE-Perf**, a new benchmark to evaluate the ability of LLMs to optimize code performance in real-world repository-level software engineering scenarios. As shown in Figure 1, the objective is to optimize the performance of a given repository codebase in the context of target functions. Each sample produces a patch that is applied to the original codebase, and the modified codebase is subsequently evaluated for performance improvements.

To construct SWE-Perf, we first extracted pull requests (PRs) from popular GitHub repositories, selecting those with strong indications of performance optimization potential. We then rigorously filtered 100K PRs by evaluating unit test runtimes on both pre-patch (original) and post-patch (modified) codebases. This process resulted in 140 data examples exhibiting observable and stable performance gains, drawn from 9 widely used GitHub repositories. Each example comprises the repository codebase, target functions, performance-related tests, the expert-authored patch, an executable environment (e.g., Docker image), and all runtime metrics. Crucially, the expert patch serves not only to confirm the feasibility of improvement but also as a human-derived gold standard against which to evaluate LLM-generated code performance optimization edits.

We conducted evaluations on several leading LLMs under two experimental settings: *oracle (file-level)* and *realistic (repo-level)*. The oracle setting assesses whether a model can optimize code performance when given relevant file contexts, whereas the realistic setting evaluates whether a model can serve as an autonomous agent capable of navigating and operating within the entire repository context without constraints. The experimental results indicate that, relative to expert performance, all LLMs exhibit significant code performance gaps, highlighting opportunities for further advancement. Furthermore, by comparing expert and model results, we analyzed the characteristics and limitations of models in handling performance enhancement tasks, thereby providing insights for further research on improving model performance. In summary, our main contributions are as follows:

- We collect the SWE-Perf dataset, the first benchmark designed to evaluate the ability of language models to optimize code performance on real-world repositories.
- We propose a repository-level performance optimization data collection pipeline, which includes a systematic method for data collection and a set of identification metrics. This framework can be easily extended to other repository-level software engineering datasets.
- We design evaluation metrics specifically for performance optimization and conduct evaluations on several LLMs under two settings. Our experimental results highlight the remaining challenges that must be addressed to meet the practical requirements of performance-aware code optimization.

2 Related Work

Code Efficiency Recent datasets targeting code efficiency, including Mercury [3], EFFIBENCH [6], EvalPerf [8], and KernelBench [15], primarily focus on function-level performance optimization. While valuable for isolated evaluation, these approaches overlook the complexity of real-world

efficiency challenges that span multiple files and modules. This oversimplification limits their ability to benchmark models' capabilities in addressing cross-cutting concerns such as dataflow refactoring or parallelism, where optimization potential is typically more substantial.

Repository-Level SWE Tasks SWE-Bench [7] first introduced a benchmark for repository-level software engineering tasks. Subsequently, related datasets including SWE-Gym [16], SWE-Lancer [11], SWE-Flow [24], and SWE-Dev [4] were developed to support various purposes, including model training and unit test evaluation. However, these datasets are primarily tailored for tasks with well-defined objectives, such as bug fixing. In contrast, efficiency optimization represents an open-ended problem lacking standardized solutions. This introduces additional complexities, including identifying optimization targets, designing performance-oriented changes, and requiring long-context understanding, planning capabilities, and efficiency-specific domain knowledge, capacities not fully addressed by existing datasets. A concurrent work, GSO [18], identifies performance-improving commits by combining an LLM-based judge with code-change heuristics, whereas our approach leverages pull requests and runtime environments for identification.

Approaches to SWE Tasks To tackle repository-level tasks, prior work has explored two major paradigms: pipeline-based, and agent-based approaches. Pipeline-based techniques, such as Agentless [20], SWE-Fixer [21], use staged workflows to solve SWE problem. Agent-based systems, like OpenHands [19], SWE-Agent [22], SWE-Smith [23] and SWE-Search [1] enable iterative reasoning across multiple steps via autonomous agents. However, these methods were not originally designed for open-ended code performance optimization, leaving room for adaptation and further exploration.

3 SWE-Perf Dataset

The SWE-Perf dataset is constructed by collecting data with performance improvement potential from popular GitHub repositories. It is designed to evaluate the capability of LLMs to optimize the performance of real-world software repositories. In the following, we provide detailed descriptions of task formulation (§3.1), data collection (§3.2), and data statistics and distribution (§3.3).

3.1 Task Formulation

As illustrated in Figure 1, the input to the SWE-Perf task consists of a codebase and a set of target functions. The output is a performance optimization patch or code, which can be applied to the original codebase to generate a new codebase.

The incorporation of target functions into task inputs is employed to restrict the evaluation scope to performance-related tests. This approach is motivated by the following considerations:

1. Running the full set of unit tests for an entire repository can be prohibitively time-consuming, which hinders the practical applicability of the benchmark. For example, in the case of the xarry repository, there are on average over 220,000 test cases, and testing just a single sample can take over one hour (on single-core CPU).

2. Given the large codebase of most repositories, there are potentially many locations where performance can be improved. Without targeted guidance, identifying and optimizing relevant regions poses significant challenges for both the model and the evaluation process.

However, as language models continue to advance and become more capable of handling full-repository optimization, we encourage future work to explore approaches that omit the performance-related unit tests and instead directly optimize the entire codebase.

3.2 Data Collection

Figure 2 illustrates the data collection process, and the specific steps of each phase are as follows:

Phase 1: Collect Pull Requests (PRs). This phase is conducted by following the methodologies of SWE-bench [7] and SWE-GYM [16]. First, we collect high-star, popular GitHub repositories. Specifically, we adopt the same 12 repositories used in SWE-bench. Second, we crawl PRs from these popular repositories. As our subsequent filtering criteria differ from those used in SWE-bench, in order to obtain more data, we re-crawl the aforementioned repositories. Third, we apply attribute filtering. In SWE-GYM and SWE-bench, two main filtering criteria are used: (1) Resolves an issue,

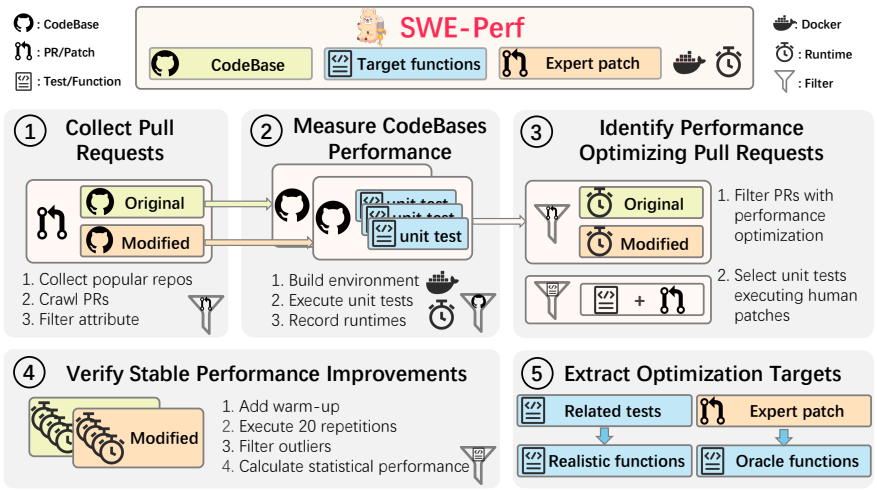

Figure 2: The data collection pipeline of our SWE-Perf benchmark. (1) collecting pull requests from popular repositories, (2) measuring unit-test performance of original and modified codebases, (3) identifying performance-optimizing pull requests, (4) verifying stable performance improvements via statistical testing, and (5) extracting optimization targets for both Oracle and Realistic settings.

and (2) Contributes tests. In this work, we retain only the first criterion. The second criterion is not adopted because our focus lies in whether the PR affects performance, specifically execution time, rather than whether it changes the correctness of unit tests. Therefore, we allow PRs that do not contribute tests, that is, PRs that do not modify unit tests.

Phase 2: Measure CodeBases Performance. Each PR collected in Phase 1 contains original and modified codebases. In this phase, we evaluate the performance of all unit tests contained in these codebases. The goal is to identify PRs that lead to performance improvements.

1. Build environment. To ensure correct execution of each codebase, we follow the approach in SWE-GYM to build a corresponding Docker image and executable Docker container for each codebase. Codebases that fail to build successfully are excluded. To ensure the comparability of performance measurements, we constrain each container to a single CPU core and 16 GB memory (5 CPU cores in Phase 4 and evaluation).

2. Execute unit tests. For each codebase, we run all unit tests inside its corresponding Docker container using pytest. This is the most time-consuming step in the entire data collection pipeline. First, the total number of codebases is large. By this stage, we have collected a total of 25,264 codebases. Second, each codebase often contains a large number of unit tests. For example, in the xarry repository, each codebase contains, on average, over 220,000 test cases, and testing a single codebase may take over one hour on a single-core CPU.

3. Record Runtimes. We use pytest to collect the execution time (runtime) of each unit test within the codebase. These runtimes serve as the basis for identifying performance changes between original and modified codebases in the subsequent phase. Codebases for which runtimes cannot be successfully collected are excluded from further analysis.

To minimize environmental effects and ensure comparability of performance measurements across codebases, we took two specific steps in this phase: 1. We standardized the execution environment by limiting each Docker container to use one CPU core and 16 GB of memory. 2. Each codebase was evaluated in three repeated experimental runs, producing three runtime measurements per unit test, which are subsequently used in Phase 3.

Phase 3: Identify PRs with Performance Optimizing Pull Requests. Based on the performance data collected in Phase 2, we identified pull requests that demonstrate significant performance improvements attributable to the associated code modifications.

1. Filter PRs with performance optimization. After Phase 2, for each PR, we have the performance (i.e., runtimes) of all unit tests in both the original and modified codebases, with each unit test measured in three repeated runs. Our goal is to identify significantly improved unit tests. PRs that do not contain such unit tests are discarded. Two filtering criteria are shown in §B.1.

2. Select unit tests executing human patches. To ensure performance improvements are attributable to the associated code modifications, we dynamically execute unit tests for each screened PR. We identify unit tests that, during dynamic execution: (1) exercise the patched code segments modified in the PR, and (2) do not execute any unit tests modified within the PR.

Phase 4: Verify Stable Performance Improvements.

Following the initial screening, we obtain a preliminary dataset. To ensure stable performance improvements, we verify results through the following procedure, retaining only unit tests whose performance gain exceeds a predefined threshold:

1. Add warm-up. To mitigate initialization-induced timing inaccuracies, before each performance measurement, we execute three performance-related unit tests to warm up the environment.

2. Execute 20 repetitions. To ensure runtime stability, each unit test is run 20 times.

3. Filter outliers. Runtime outliers within the 20 measurements are identified and removed using the Interquartile Range (IQR) method with a threshold multiplier of 1. Specifically, let $Q1$ and $Q3$ represent the first and third quartiles of the runtime sample, respectively. The interquartile range is $IQR = Q3 - Q1$. Data points r_i satisfying either condition below are classified as outliers and removed:

Algorithm 1: Compute Statistically Significant Minimum Performance Gain (δ)

Input : $A = [a_1, a_2, \ldots, a_n]$: Filtered runtimes (modified version)
$B = [b_1, b_2, \ldots, b_m]$: Filtered runtimes (original version)
α : Significance level (default = 0.1)
$step$: Gain increment step (default = 0.01)
max_x : Maximum gain to test (default = 1.0)
Output : δ : Conservative minimum significant performance gain

$x \leftarrow 0.0$
$\delta \leftarrow 0.0$
while $x \leq max\_x$ **do**
  $B_{\text{adj}} \leftarrow B \times (1 - x)$ `// Pessimistically weaken improvement`
  $p \leftarrow$ MannWhitneyUTest(B_{adj}, A, alternative = 'greater')
  if $p < \alpha$ **then**
    $\delta \leftarrow x$ `// Update conservative gain`
    $x \leftarrow x + step$ `// Test next larger gain`
  else
    break `// Significance lost, stop searching`
  end
end
return δ

$$r_i < Q1 - k \times IQR, \;\; r_i > Q3 - k \times IQR \tag{1}$$

where, $k = 1$ (the threshold multiplier).

4. Calculate statistical performance. To confirm stable performance improvement, we compute a statistically significant minimum performance gain (δ) for each test case using Algorithm 1. Here, δ denotes the largest value such that the modified runtime distribution remains significantly faster than the original under conservative adjustments (Mann-Whitney U test, $p < p\_threshold$ (0.1). Unit tests with δ exceeding the threshold (0.05) constitute the final SWE-Perf dataset.

Phase 5: Extract Optimization Targets. Following the aforementioned four phases, we have filtered PR data and associated unit tests that demonstrate stable performance improvements. In this phase, we extract the target functions for model optimization. We categorize the task into two settings and extract target functions accordingly: Oracle and Realistic.

1. Oracle (File-Level): This setting aims to provide the model with the oracle functions as the optimization target and the related entire files as contextual information, evaluating the model's capability to generate purely performance-enhancing code. The target functions are directly modified. We extract this target functions from the human patch in the PR by combining AST (Abstract Syntax Tree) analysis with unified diff matching.

2. Realistic (Repo-Level): This setting simulates an end-to-end real-world scenario, providing the system (e.g., OpenHands Agent) with the functions measured during testing (unit test execution) as the optimization target. The system has greater freedom to modify code across the entire repository, measuring the system's ability to enhance performance repository-wide. The target functions are the directly measured functions, not necessarily the ones directly modified, as improvements may involve functions they call. Compared to the Oracle setting, the Realistic setting is more challenging, requiring both performance-improving code generation and additional capabilities such as retrieval.

We identify the target functions by using yappi to record functions dynamically executed during performance-related unit tests. Combined with AST parsing, we determine the specific functions directly invoked by the unit test. We avoid using the unit test itself as the target to prevent test information leakage, which could lead the model to perform functional pruning – modifying the code to retain only the functionality exercised by the test and solely to meet the optimization metric.

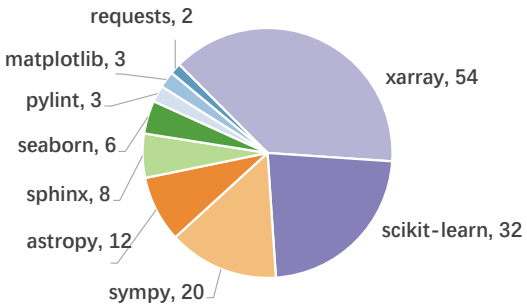

Figure 3: Distribution of SWE-Perf across 9 open source popular GitHub repositories.

Table 1: Average and maximum values of various SWE-Perf attributes.

Category	Metric	Mean	Max
Size	# Instances	140	
	# Repos	9	
Codebase	# Files (non-test)	447.3	1,972
	# Lines (non-test)	170k	502k
Expert Patch	# Lines edited	131.1	1,967
	# Files edited	4.3	71
	# Func. Edited	7.6	94
Tests	# Related tests	8.1	68
	Original runtime / s	0.28	25.2
Functions	# Oracle	7.6	94
	# Realistic	30.1	256
Performance	Ratio	10.9%	87.8%

After the aforementioned five phases, we transform them into the SWE-Perf dataset, which consists of the following components:

1. **CodeBase**: The source code corresponding to the original codebase.

2. **Executable Environment**: The Docker image and container used to execute the original codebase.

3. **Target Functions**: The optimization-target functions categorized as oracle and realistic.

4. **Performance-related Unit Tests**: The unit tests identified as having performance improvements.

5. **Runtime Metrics**: Original and modified codebase runtime metrics for performance-related tests.

6. **Expert Patch**: The expert patch from the modified codebase. These serve as references for human-level performance optimization.

3.3 Data Statistics and Distribution

In Phase 1, a total of 102,241 pull requests were collected, from which 19,797 pull requests remained after filtering. In Phase 2, 34,397 distinct codebases were gathered, and test executions were successfully performed on 19,499 of them, yielding corresponding runtime data. In Phase 3, 4413 PRs were identified whose main and dev codebases both had available runtime data, from which 1,696 valid instances were derived. If Phase 4, 140 instances were identified. Detailed statistics and data distributions of SWE-Perf are presented in Figure 3 and Table 1.

4 Evaluation Methodology

We designed a three-level performance evaluation framework with three progressively stringent metrics: *Apply*, *Correctness*, and *Performance*. During evaluation, we first apply the model-generated patch/code to the original codebase ($codebase\_pre_i$) to obtain the post-patch codebase ($codebase\_post_i$). Subsequently, within the corresponding Docker environment, we execute all performance-related tests ($test_{i,j}, j \in \{1, \ldots, N_i\}$) on both the original and post-patch codebases, collecting the results ($result\_pre_{i,j}, result\_post_{i,j}$) and runtime ($runtime\_pre_{i,j}, runtime\_post_{i,j}$) measurements for comparison.

It is worth noting that, to eliminate the impact of environmental variability on execution speed, we re-evaluated the original codebase runtime during the testing phase, even when the original codebase runtime from data collection was available. This ensures full comparability between the original and post-patch codebase runtime measurements.

The definitions and corresponding formulas for the three evaluation metrics are as follows:

Apply: This metric evaluates whether the model-generated patches or code can be successfully applied to the original codebase without conflicts or errors. $Apply = \frac{N_{apply}}{N_{total}}$, where, N_{apply} is the number of successfully applied samples, N_{total} is the number of total samples.

Correctness: This metric assesses whether the patch preserves the functional correctness of the code, specifically whether all unit tests pass successfully after the patch is applied.

Table 2: Experimental results of leading LLMs under Oracle and Realistic settings on SWE-Perf.

Setting	Methods	Apply	Correctness	Performance
	Expert	100.00%	100.00%	10.85%
Oracle (File-Level)	Claude-3.7-sonnet	66.43%	61.43%	1.24%
	Claude-4-sonnet	73.57%	70.00%	1.76%
	Claude-4-opus	85.71%	78.57%	1.28%
	GPT-4o	63.57%	56.43%	0.60%
	OpenAI-o1	66.42%	63.57%	0.41%
	OpenAI-o3	78.57%	76.43%	1.37%
	DeepSeek-V3	47.85%	42.86%	0.54%
	DeepSeek-R1	55.71%	51.43%	0.90%
	Gemini-2.5-Pro	**95.00%**	**83.57%**	1.48%
	Qwen3-235B-A22B	54.29%	48.57%	0.68%
Realistic (Repo-Level)	Claude-3.7-sonnet (Agentless)	88.57%	70.71%	0.41%
	Claude-3.7-sonnet (OpenHands)	87.86%	77.86%	**2.26%**

$$Correctness = \frac{\sum_{i=1}^{N_{total}} \left[\bigwedge_{j=1}^{N_i} result\_post_{i,j} = pass \right]}{N_{total}}$$

where $result\_post_{i,j}$ is the post-patch result of the j-th unit test on the i-th sample. N_i is the number of performance-related unit tests for the i-th sample. $\bigwedge$ is the logical AND for all tests j, indicating the sample must pass every test.

Performance: This metric measures the statistically significant minimum performance gain introduced by the patch, based on runtime comparisons. The computation process resembles that of Phase 4 in data collection (§3.2). After a warm-up period, 20 repetitions and outliers filtering, the Minimum Performance Gain ($p_{i,j}$) for each instance i and each unit test j is calculated using Algorithm 1. Performance is then calculated as follows:

$$Performance = \frac{1}{N_{total}} \sum_{i=1}^{N_{total}} P_i, \;\; P_i = \frac{1}{n_i} \sum_{j=1}^{N_i} p_{i,j} \tag{2}$$

5 Experiments

This section presents the baseline setting (§5.1), the main experimental results (§5.2), and a further analysis of the model's performance (§5.3).

5.1 Baselines

Recent work on repository-level software engineering tasks (*e.g.*, SWE-Bench) can be broadly categorized into three paradigms: direct model approaches, pipeline-based methods and agent-based systems. We select representative state-of-the-art (SOTA) methods from each category for evaluation.

1. Oracle: For the oracle setting, we adopt a chain-of-thought prompting strategy to directly use model to enhance codebase performance. The model is provided with the oracle files extracted from expert patches and oracle target functions. A single-pass inference is used to generate the patch. We evaluate with 10 popular models.

2. Agentless (Pipeline-Based): Agentless [20] follows a pipeline-based approach to address the task. It employs a fixed multi-stage workflow consisting of hierarchical fault localization, code repair, and candidate patch selection through regression and reproduction testing.

3. OpenHands (Agent-Based): OpenHands [19] adopts an agent-based methodology. It provides a flexible and extensible platform for building autonomous software development agents, enabling iterative reasoning and interaction across multiple steps in the software engineering process.

For both Agentless and OpenHands, we use Claude-3.7-sonnet as the base model. Because Claude-3.7-sonnet is the officially recommended backend for OpenHands [3], as it has been reported to work best within OpenHands. The implementation details are provided in Appendix §C.

5.2 Main Results

The performance results of various methods on SWE-Perf are presented in Table 2. The comparative performance across different repositories is illustrated in Figure 4. From these empirical findings, we derive the following conclusions:

[3] `https://github.com/All-Hands-AI/OpenHands`

Compared to the expert, all models exhibit substantial room for improvement on SWE-Perf. OpenHands demonstrates superior performance due to its agent-based methodology, providing a flexible and extensible platform for autonomous software development agents that enable iterative reasoning and multi-step interaction throughout the software engineering process. However, a significant performance gap of 8.59% persists between OpenHands and Expert, indicating considerable potential for further.

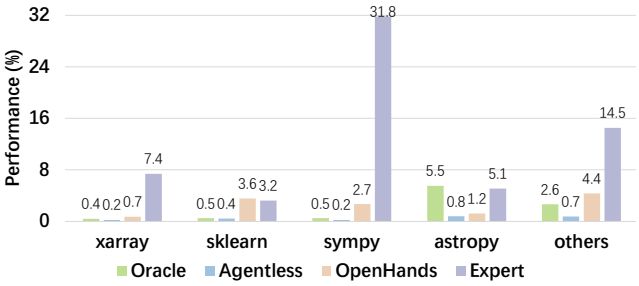

Figure 4: Performance for different methods across the 9 repositories represented in SWE-Perf. The base model used in the Oracle setting is Claude-3.7-sonnet.

The model exhibits the potential to surpass expert performance and achieve superior results. As can be observed in Figure 4, the model already rivals the performance of the Expert on certain repositories; for instance, on sklearn, OpenHands outperforms the Expert by 0.4%. This demonstrates that the model can achieve a competitive edge over established expert methods in specific tasks or datasets, signifying an early-stage breakthrough.

5.3 Analysis of Model Capabilities

This subsection analyzes performance from four aspects: (1) isolating performance from correctness metrics, (2) quantifying target functions' impact, (3) evaluating runtime-performance relationships, and (4) identifying keyword patterns (§5.4). We aim to uncover optimization bottlenecks and actionable improvement insights.

5.3.1 Performance Analysis Decoupled from Correctness

To decouple the model's code performance enhancement capability from its code/patch generation correctness, we calculated performance metrics exclusively for correct examples by modifying the denominator in Equation (2) from N_{total} to $N_{correctness}$. The results are presented in Figure 6 and Figure 5. The Expert metric represents expert performance calculated using only correct samples from the corresponding method.

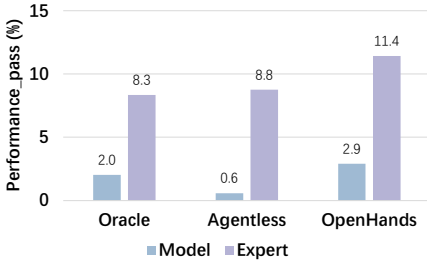

Figure 6: Performance for correct examples. The expert performance calculated using only correct examples from the corresponding method.

OpenHands demonstrates superior performance particularly excelling in scenarios with higher potential performance ceilings. As illustrated in Figure 6, using identical models, OpenHands achieves an approximately 3% higher benchmark performance compared to alternative methods, highlighting its superior capability in translating model capacity into realized performance, especially near the achievable ceiling.

5.3.2 Impact of Runtime on Performance

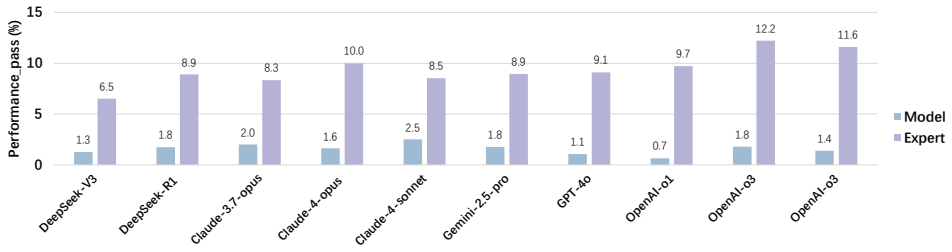

Figure 5: Performance for correct examples. The expert performance calculated using only correct examples from the corresponding method.

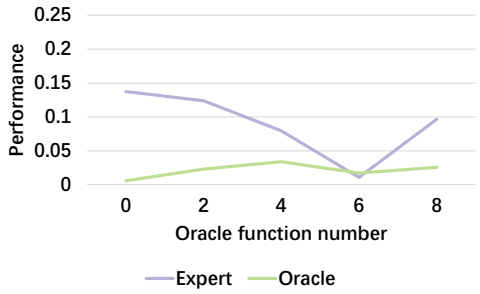

Figure 8: Performance variation relative to the number of Oracle target functions.

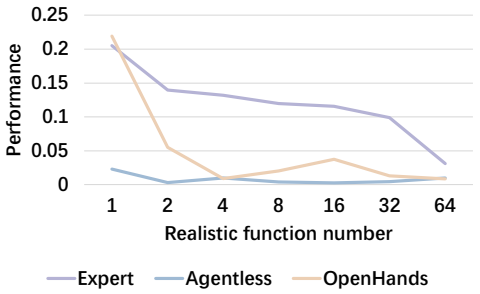

Figure 9: Performance variation relative to the number of Realistic target functions.

To examine the model's ability to improve performance across runtimes, we plotted the chart shown in Figure 7.

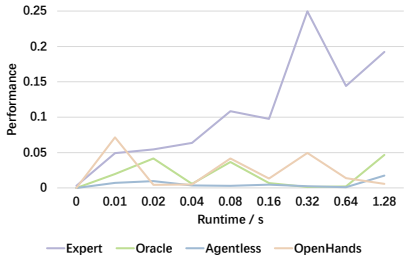

Figure 7: Performance variation relative to the runtime of the original codebase.

As runtime increases, the performance ceiling rises correspondingly. As evidenced in Figure 7, the expert model's performance demonstrates a progressive upward trend with extended runtime, indicating that longer computation times enable more sophisticated optimization and convergence towards higher performance potentials.

The model's capability to improve performance on cases with longer runtimes requires further enhancement. Figure 7 reveals that while expert performance continues to climb with increased runtime, the model's performance plateaus (or remains stagnant). This divergence underscores the critical need to analyze and emulate the expert's optimization strategies specifically under extended-runtime conditions to enhance the model's performance scalability.

5.3.3 Impact of Target Functions on Performance

To investigate the impact of the number of target functions on model performance, we present the statistics summarized in Figure 8 and Figure 9.

As the number of target functions increases, performance improvements become increasingly difficult to achieve. Figure 9 shows that the performance of experts declines with the addition of functions, indicating heightened difficulty in enhancing performance and a lower performance ceiling.

The model should prioritize learning from the expert to enhance its capability when handling a larger number of target functions. Figure 9 reveals that OpenHands achieves performance comparable to the expert at lower function counts. However, the performance gap widens significantly as the number of functions increases. Future research aimed at improving model performance should focus on optimizing for multi-function scenarios.

5.4 Keyword Analysis for Performance

To further investigate the differences in modification strategies preferred by the model and the expert when enhancing performance, we respectively generated word clouds representing the lines added in patches, as shown in Figure 10 and Figure 11.

OpenHands patches focus on low-level data structures and basic functionality. The word cloud reveals frequent terms such as "children," "identifier," "time," and "attributes," indicating that the changes are centered on refining structural components and enhancing fundamental operations. These optimizations likely address data management and attribute handling, focusing on the internal mechanics of the code.

Expert patches emphasize high-level abstractions and data integrity. The presence of terms like "literal," "value," "type," "label," and "dtype" suggests that the optimizations are geared towards improving type safety, type annotations, and handling of data values. These changes likely aim

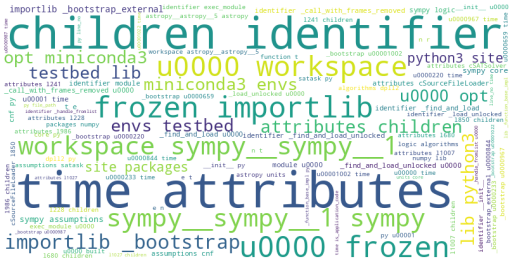

Figure 10: Word cloud of lines added in OpenHands patches.

Figure 11: Word cloud of lines added in Expert patches.

to enhance error handling, code clarity, and overall system efficiency by addressing more abstract elements of the code.

The model-generated patches exhibit a strong focus on foundational infrastructure and low-level operations, as indicated by frequent terms such as "miniconda3", "envs", "frozen", "importlib", and "bootstrapu0000". This suggests an emphasis on environment configuration, dependency management, and basic module handling. The presence of encoded fragments ("u0000") further implies automated generation targeting syntactic adjustments or toolchain compatibility, rather than semantic, application-level optimization.

The expert patches demonstrate a clear orientation towards domain-specific functionality and performance-critical enhancements. Key terms like "sympy", "time", "workspace", and "active packages" reveal a deliberate strategy to optimize core computational workflows (e.g., symbolic mathematics with sympy), resource management (workspace), and runtime efficiency (time). This reflects human expertise in restructuring high-level components to improve performance, maintainability, and domain-relevant operations.

6 Conclusion

In conclusion, SWE-Perf addresses a critical gap in current benchmarking by providing the first repository-level dataset focused on realistic code performance optimization, a task traditionally reliant on human expertise and largely unexplored in prior LLM evaluations. Our benchmark and comprehensive baseline assessments reveal substantial room for improvement in current models, underscoring the complexity of cross-module and repository-scale optimizations in real-world software. The significant performance gaps observed between existing LLMs and expert-level optimization highlight the need for novel approaches that can handle the intricacies of repository-level performance enhancement. By establishing this new standard and evaluation framework, we pave the way for future research to advance the capabilities of language models in delivering meaningful, performance-aware code enhancements at scale, ultimately bridging the gap between automated code generation and expert-level optimization in production environments.

References

[1] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement, 2025.

[2] Cursor. The ai code editor. `https://www.cursor.com/en`, 2025. Accessed: 2025-05-12.

[3] Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37, 2024.

[4] Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development. 2025.

[5] Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. Codeeditorbench: Evaluating code editing capability of large language models. *CoRR*, abs/2404.03543, 2024.

[6] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. Effibench: Benchmarking the efficiency of automatically generated code. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 11506–11544. Curran Associates, Inc., 2024.

[7] 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 *The Twelfth International Conference on Learning Representations*, 2024.

[8] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. *CoRR*, abs/2408.06450, 2024.

[9] Javier Mancebo, Félix García, and Coral Calero. A process for analysing the energy efficiency of software. *Information and Software Technology*, 134:106560, 2021.

[10] Microsoft. Ai that builds with you. `https://github.com/features/copilot`, 2025. Accessed: 2025-05-12.

[11] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $ 1 million from real-world freelance software engineering?, 2025.

[12] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering? *CoRR*, abs/2502.12115, 2025.

[13] Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[14] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. Artificial intelligence vs. software engineers: An empirical study on performance and efficiency using chatgpt. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, CASCON '23, page 24–33, USA, 2023. IBM Corp.

[15] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.

[16] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024.

[17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[18] Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: challenging software optimization tasks for evaluating swe-agents. *CoRR*, abs/2505.23671, 2025.

[19] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents, 2024.

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

[21] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*, 2025.

[22] 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 *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[23] John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *CoRR*, abs/2504.21798, 2025.

[24] Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Swe-flow: Synthesizing software engineering data in a test-driven manner. *CoRR*, abs/2506.09003, 2025.

Appendix

A Limitations and Safeguards:

A.1 Limitations

Our work has two primary limitations. First, the current version of SWE-Perf is constructed from a limited set of open-source repositories, future work could expand the dataset to improve coverage and generalizability. Second, while we use human-written patches as ground truth to assess model performance, these may not represent the optimal achievable performance, potentially underestimating the true upper bound of improvement.

A.2 Safeguards

To ensure the responsible use of SWE-Perf, we implement the following safeguards. All codebases included in the dataset are drawn from permissively licensed open-source projects, and we exclude any repositories containing sensitive or personal information. We provide executable environments in sandboxed Docker containers to prevent unsafe code execution. Moreover, model-generated patches are evaluated automatically in isolated environments, reducing the risk of harmful or unintended system behavior. The dataset and code will be released under an academic research license, and users will be required to agree to the terms of use restricting deployment in sensitive or safety-critical systems.

B Dataset

All experiments were conducted on two Linux machines. Each machine is equipped with 256 logical CPU cores (128 physical cores across 2 sockets, with 2 threads per core) and 2.0 TiB of RAM.

B.1 Details of Data Collections

There are two filtering criteria in Phase 3:

(1) Correctness: The unit test must pass in both the original and modified codebases, i.e., the pytest result must be "pass" in both cases.

(2) Performance Ratio: To ensure that the improvement is substantial in practical terms, we compute the optimized ratio of the modified to original runtimes. The optimized ratio is computed per unit test per pull request as the mean of three experimental replicates. The ratio must be above a specified threshold (0.3). The ratio is calculated as:

$$Ratio = \frac{R_{original} - R_{modified}}{R_{original}}$$

where, R is the runtime for each unit test.

B.2 Details of Data Statistics

The statistical summary of data volume at each stage is presented in Table 3.

Table 4 presents the runtime statistics for each codebase in Phase 2 - Step 2.

The mapping table of repository abbreviations to their full names is provided in Table 5.

C Baselines

C.1 Details of Baselines

1. Oracle: The specific prompts used for the Oracle is shown in Figure 14.
   (a) OpenAI/GPT: The model version used is o1-preview-2024-09-12, o3-2025-04-16, gpt-4o-2024-11-20, with a temperature of 0.2, top-p of 0.1, and a maximum token limit of 8192.

Table 3: Table of data volume at each phase.

Repo	Phase1			Phase2		Phase3			Phase4
	PRs	Tasks	Tasks with versions	Unique codebase	Codebase with pytest report	Tasks with success runtimes	Tasks after step1	Tasks after step2	Tasks
astropy	11555	1947	1947	3589	2323	529	409	195	12
django	13200	4179	4179	6887	6399	-	-	-	-
matplotlib	18791	2576	2576	4707	1648	495	369	163	3
seaborn	1115	304	304	573	539	199	165	76	6
flask	2637	285	262	521	410	62	0	-	-
requests	2507	223	217	410	364	151	105	43	
xarray	4470	1359	1354	2485	1000	517	495	327	54
pylint	4553	1174	911	1680	1522	806	772	405	3
pytest	6228	1203	989	1860	1717	583	397	0	2
sklearn	18079	2576	2574	4721	431	181	181	94	32
sphinx	6002	1507	1385	2629	2299	630	573	236	8
sympy	13104	2464	2464	4335	847	260	254	157	20
sum	102241	19797	19162	34397	19499	4413	3720	1696	140

Table 4: Runtime statistics for each codebase in phase 2 - step 2. Only runtimes from successful pytest executions are included. All time measurements are in minutes. **Test runtime** refers to the time taken to execute pytest for an individual codebase, while **Total duration** denotes the overall execution time for a single codebase, including operations such as Docker image building and pytest execution.

Repo	Test runtime		Total duration	
	avg	max	avg	max
astropy	3.09	21.57	4.36	24.12
django	0.22	0.91	1.52	5.85
matplotlib	7.60	16.96	10.69	28.75
seaborn	3.37	14.56	4.15	15.97
flask	0.04	0.10	1.04	1.75
requests	1.09	9.69	1.88	11.08
xarray	58.11	119.95	58.52	121.03
pylint	2.75	3.99	3.47	6.13
pytest	2.34	18.94	3.13	20.10
sklearn	83.89	119.96	85.83	125.98
sphinx	8.72	38.99	9.57	40.00
sympy	24.60	112.17	24.98	112.57

Table 5: Mapping table of repository abbreviations to full names.

Repo name	Repo full name
astropy	astropy/astropy
matplotlib	matplotlib/matplotlib
seaborn	mwaskom/seaborn
requests	psf/requests
xarray	pydata/xarray
pylint	pylint-dev/pylint
sklearn	scikit-learn/scikit-learn
sphinx	sphinx-doc/sphinx
sympy	sympy/sympy

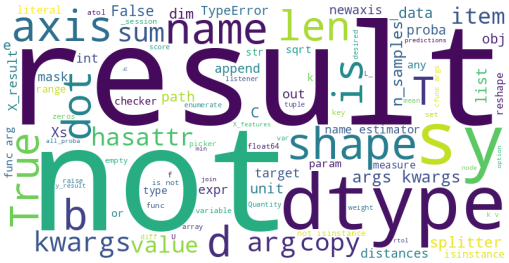

Figure 12: Word cloud of lines added in Agentless patches.

Figure 13: Word cloud of lines added in Oracle(Claude-3.7) patches.

(b) Claude: The model version is gcp-claude37-sonnet, gcp-claude4-opus, gcp-claude4-sonnet. The thinking feature is enabled, with a thinking budget of 2000 tokens and a maximum token output of 8192.

(c) DeepSeek: The versions used are deepseek-r1-0528 and DeepSeek-V3.

(d) Gemini: The versions used are gemini-2.5-pro-preview-05-06.

(e) Qwen: The versions used are Qwen3-235B-A22B.

2. Agentless [4]: The sample number is set to 1.

3. OpenHands [5]: The maximum number of iterations is set to 50.

C.2 Details of Word Cloud

More comprehensive word clouds are presented in Figure 12, Figure 13.

[4] https://github.com/OpenAutoCoder/Agentless

[5] https://github.com/All-Hands-AI/OpenHands

Oracle prompt:

You will be provided with a partial code base and objective functions. You need to improve the objective function's efficiency and execution speed by editing the code base.
<problem_statement>
Please enhance the computational efficiency and execution speed across the entire repository. The optimization efforts may target one or more objective functions, including but not limited to:
[target_functions]
The following conditions apply:
1. Acceleration of at least one objective function is sufficient for success, as performance evaluations will be conducted collectively on all targeted functions.
2. Optimization may be achieved either directly through modifications to the objective functions or indirectly by improving computationally intensive subroutines upon which they depend.
3. Optimization efforts should prioritize maximal efficiency gains where feasible.
4. All existing unit tests must remain unaltered to preserve functional correctness.
</problem_statement>

<code>
[content_of_files]
</code>

Please improve its efficiency and execution speed by generate *SEARCH/REPLACE* edits to fix the issue.

Every *SEARCH/REPLACE* edit must use this format:
1. The file path
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE
7. You can't edit the test case, only the code base.
8. Only use standard python libraries, don't suggest installing any packages.

Here is an example:

```python
### mathweb/flask/app.py
<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
>>>>>>> REPLACE
```

Please note that the *SEARCH/REPLACE* edit REQUIRES PROPER INDENTATION. If you would like to add the line ' print(x)', you must fully write that out, with all those spaces before the code!
Wrap the *SEARCH/REPLACE* edit in blocks ```python...```.

Figure 14: Oracle prompt.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Our experiments correspond perfectly to the contributions mentioned in our introduction.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations of our work in the Appendix.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: We have provided assumptions for the relevant theoretical reasoning in the Dataset and Evaluation sections.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The results of our experimental section are all reproducible.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Once accepted, all data and code mentioned in the paper will be made publicly available.

   Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: In the "Experiment" and "Appendix" section of the main text, we furnish comprehensive information regarding the experimental parameters and settings.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: It is discussed in "Dataset" section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).

- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We offer the detailed information about computer resources in the implement details part of the "Appendix".

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This paper adheres strictly to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: There is no societal impact of our work performed.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [Yes]

    Justification: We discuss the Safeguards of our work in the Appendix.412

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The models used in our paper are available and have been licensed for use.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: SWE-Perf dataset is documented in "Datasets" section.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.

- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: We do not involve crowd-sourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: We do not involve crowd-sourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Our method uses LLM as baselines, with detailed implementation described in the main text.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.