The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason

Shanchao Liang[1], Spandan Garg[2], Roshanak Zilouchian Moghaddam[2]
[1]Purdue University
[2]Microsoft

Abstract

As large language models (LLMs) become increasingly capable and widely adopted, benchmarks play a central role in assessing their practical utility. For example, SWE-Bench Verified has emerged as a critical benchmark for evaluating LLMs' software engineering abilities, particularly their aptitude for resolving real-world GitHub issues. Recent LLMs show impressive performance on SWE-Bench Verified, leading to optimism about their capacity for complex coding tasks. However, current evaluation protocols may overstate these models' true capabilities. It is crucial to distinguish LLMs' generalizable problem-solving ability and other learned artifacts. In this work, we introduce two new diagnostic tasks: *file path identification* from issue descriptions alone and *ground truth function reproduction* with only the current file context and issue description to probe models' underlying knowledge. We present empirical evidence that performance gains on SWE-Bench Verified may be partially driven by memorization rather than genuine problem-solving. We show that state-of-the-art (SoTA) models achieve up to 76% accuracy in identifying buggy file paths using only issue descriptions, without access to repository structure. This performance is merely up to 53% on tasks from repositories not included in SWE-Bench, pointing to possible data contamination or memorization. Similar patterns are also observed for the function reproduction task, where the verbatim similarity is much higher on SWE-Bench Verified than on other similar coding benchmarks (up to 35% consecutive 5-gram overlap ratio on SWE-Bench Verified and Full, but only up to 18% for tasks in other benchmarks). These findings raise concerns about the validity of existing results and underscore the need for more robust, contamination-resistant benchmarks to reliably evaluate LLMs' coding abilities.

1 Introduction

As Large Language Models (LLMs) become increasingly integrated into software development tools and workflows, the need for rigorous evaluation of their coding capabilities has grown significantly Srivastava et al. [2023]. Standardized benchmarks play a central role in this evaluation, offering reproducible metrics across diverse tasks Zhuo et al. [2024], Pan et al. [2024], Bowman and Dahl [2021]. Among these, SWE-Bench Verified has emerged as a prominent benchmark for assessing LLMs' ability to resolve real-world GitHub issues Jimenez et al. [2024], OpenAI [2024]. Recent advances have shown LLMs, e.g., Tools + Claude 4 Opus, are achieving over 70% Pass@1 accuracy on SWE-Bench, suggesting substantial progress Anthropic [2025].

However, the rapid improvement in benchmark performance raises a fundamental question: to what extent do these gains reflect genuine, generalizable problem-solving abilities versus memorized patterns from training data? This concern is particularly acute because training corpora for LLMs often include the same public code repositories from which benchmarks like SWE-Bench are constructed. For instance, GitHub repositories are prominently featured in training datasets for LLaMA Touvron et al. [2023], PaLM Chowdhery et al. [2022], and Codex Chen et al. [2021], creating potential overlap between training and evaluation data.

To investigate this question systematically, we focus on two critical components of software engineering tasks: bug localization, the ability to identify which files contain bugs, and patch generation, the ability to generate the corrected version of the buggy code. Both sub-tasks are fundamental to successful issue resolution. For bug localization Hossain et al. [2024], models should require a genuine understanding of both the problem description and codebase structure to identify relevant files. For patch generation, models need to understand the semantics of the code and, based on the issue description to reason and generate the correct solution. Both capabilities could be compromised by memorization: models might recall specific issue-file associations or reproduce memorized code patterns rather than demonstrate genuine problem-solving ability. To systematically investigate these different memorization patterns, we design three diagnostic tasks: file-path identification, function reproduction, and prefix completion. The file-path identification task requires models to identify buggy files using only GitHub issue descriptions, deliberately withholding all repository structure and code context. The function reproduction task measures models' ability to reproduce fixed function code without providing the necessary information for such reproduction, such as the respective function specification. The prefix completion task checks models' ability to generate solutions verbatim from task prefixes.

However, directly measuring memorization versus genuine reasoning is challenging, as models legitimately learn coding patterns during training, and there is a lack of a standard baseline for their understanding of general coding abilities. To address this challenge, we propose a cross-benchmark performance analysis approach and use comparative analysis across similar systems as proxies. In our case, we evaluate models across multiple similar benchmarks and repositories, using performance differences across similar tasks as indicators of memorization. Genuine coding skill should lead to similar performance across comparable tasks, while memorization would result in unusually high scores on tasks that likely overlap with the model's training data.

Our results reveal concerning patterns: on the file-path identification task, SoTA models like o3 **?** achieve up to 76% accuracy on SWE-Bench-Verified instances, despite lacking the contextual information that should be necessary for this task. More critically, we observe substantial performance drops when evaluating on external benchmarks and repositories, with both file-path accuracy and verbatim similarity showing significantly higher values on SWE-Bench Verified compared to other evaluation sets, suggesting benchmark-specific memorization rather than generalizable coding proficiency.

These findings raise serious concerns about the validity of current benchmark evaluations in software engineering and highlight the need for more robust, contamination-resistant evaluation approaches.

Our contributions include:

1. **Cross-Benchmark Performance Analysis for Memorization Detection**: We introduce a systematic approach for benchmark memorization detection using performance disparities across similar benchmarks. When models exhibit disproportionately high performance on specific benchmarks compared to similar tasks, this indicates potential training data contamination rather than genuine coding ability.

2. **Diagnostic Task Design**: We introduce two controlled diagnostic tasks: file-path identification without repository context and verbatim patch reproduction analysis that isolate and measure different types of memorization effects in software engineering benchmarks.

3. **Evidence of Benchmark Compromise**: We show that LLMs achieve suspiciously high performance on SWE-Bench (76% context-free accuracy, elevated 5-gram overlap) but show a drop on external benchmarks, indicating memorization rather than genuine capability.

2 Approach

In this section, we introduce the three diagnostic tasks designed for the detection of memorization in software engineering benchmarks: file path identification tests whether models can locate buggy files without repo context, function reproduction checks complete implementation generation without specifications, and prefix completion directly measures verbatim code sequence reproduction.

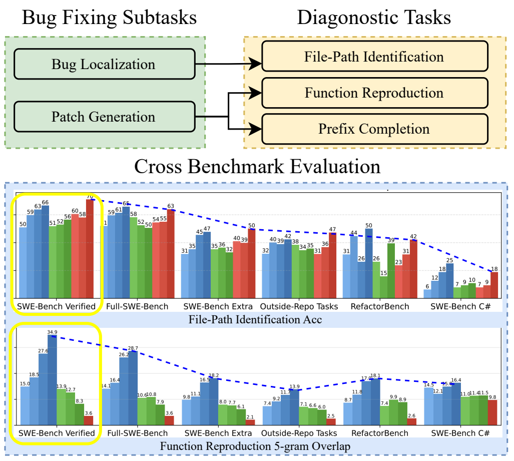

Figure 1: Overview of our benchmark memorization detection approach.

File Path Identification Prompt Template

```
You will be provided with an issue
statement explaining a problem to resolve
within a codebase. The code base is:
scikit-learn/scikit-learn.
<issue>
Issue description here
</issue>
...
DISCUSSION
Discuss here with yourself about how you came
up with this response.
RESPONSE
‘‘‘
path/to/buggy/file.py
‘‘‘
```

Figure 2: The prompt template used to test models' ability to identify buggy files without repo access.

2.1 File Path Identification Task

In this task, models are presented with issue reports and asked to predict the file path that would require modification to resolve the described issue. Importantly, models do not receive access to repo structure, code, or metadata; only the issue text and repo name are provided. The model's prediction is considered correct if it exactly matches one of the file paths in the ground truth (GT) patch. By narrowing down the scope to file identification and removing the repo context provided to the LLMs, we aim to expose memorization artifacts apart from their problem-solving abilities. In addition, we recognize that issue descriptions themselves may contain file paths. To resolve this, we also report a filtered accuracy metric on instances where such explicit path mentions are not observed; details available in Section 3.3.

Figure 2 illustrates the prompt template used for this task. Each prompt includes: **(1) the name of the repo** and **(2) the issue description**. The model is instructed to predict a single file path that would need to be modified to resolve the issue. Additionally, the model is asked to provide its reasoning, which enables us to analyze its underlying thought process. We show the prompt template in Figure 2.

2.2 Function Reproduction Task

In the Function Reproduction Task, we analyze whether models can generate complete functions that are modified by the GT patch without knowing its specification or function signature. Specifically, we present models with issue descriptions alongside buggy files where functions modified by the GT patch are completely removed. Models must generate complete implementations for each missing function. Since the function specification is unavailable, successful reproduction indicates memorization of specific textual sequences rather than problem-solving ability. We compare performance across benchmarks to isolate memorization effects from general programming ability. The prompt template is available in the supplementary materials.

Our approach directly tests memorization by requiring models to reproduce complete function implementations without access to signatures, docstrings, or relevant context from other files. We note that this is a single round generation, and the models must rely only on the provided file content to reproduce the removed functions. The models do not have access to the repo. These constraints ensure that successful reproduction requires prior exposure to the specific implementation rather than general problem-solving ability.

2.3 Prefix Completion Task

Additionally, we designed a controlled experiment testing whether models can reproduce exact code sequences in the task solution given only the prefix code. The task proceeds in three phases:

Prefix Extraction: For each instance in SWE-Bench Verified, we extract the lines of code before each buggy code snippet that was modified by the GT patch.

Prompt Construction: The extracted code prefix is provided to the model, along with instructions to complete the remaining code implementation.

Generation and Evaluation: Models generate continuations based solely on the provided prefix. We then compare the generated code against the GT to identify verbatim matches for up to N lines of code. N is the number of lines that are modified by the GT patch at the respective location.

This Prefix Completion Task complements the Function Reproduction Task by testing memorization in a setting that directly mirrors the models' auto-regressive pre-training. Together, these two tasks assess memorization at two different scales: the recall of complete functions versus the verbatim completion of sequential code.

3 Experiment Setup

In this section, we introduce the benchmarks and metrics used to evaluate the models.

3.1 Benchmarks

SWE-Bench-Verified is a human-validated subset of the original SWE-Bench dataset, consisting of 500 curated samples from 12 open-source Python repositories. Each task presents a GitHub issue description and requires generating code edits to resolve it. In standard settings, models are given access to the repo and the issue description. However, in our experiments, we provide only context given by the task as detailed in Section 2.

SWE-Bench-C# is an internal benchmark that follows SWE-Bench's construction pipeline and comprises 75 tasks from eleven C# repositories. Each task has the same format as SWE-Bench, requiring code edits based on a given issue description. This benchmark allows us to test whether performance patterns observed on Python-based tasks extend to tasks from different programming languages, helping to distinguish between language-specific memorization and genuine coding capabilities. Details are available in the supplementary materials.

RefactorBench Gautam et al. [2025] is a benchmark of 100 handcrafted, multi-file refactoring tasks from 9 open-source Python repos. Each task presents natural language instructions and requires the model to generate a patch that performs the described refactoring operation. For our experiment, we use a subset of 39 tasks from the full dataset. Unlike SWE-Bench's focus on bug fixes, RefactorBench tests code restructuring abilities across different repos and task types. If models show significantly different performance patterns on RefactorBench compared to SWE-Bench despite both being Python-based, this could indicate task or repo-specific optimization rather than general coding proficiency.

Table 1: Average number of files and issue length per benchmark.

Benchmarks	# of Files	Issue Length
SWE-Bench Verified	763.5	451.2
SWE-Bench C#	716.7	586.2
RefactorBench	1149.2	14.6

Additionally, to ensure fair comparison across benchmarks and verify that performance differences reflect contamination rather than varying task difficulty, we analyze two key factors that may influence path identification complexity. In Table 1, we compute the average number of files with target extensions (.py for Python, .cs for C# repositories) as a proxy for search space complexity and average issue description length as a measure of available contextual info. SWE-Bench-Verified and SWE-Bench-C# have similar repo sizes (763.5 vs 716.7 files) and comparable issue lengths (451.2 vs 586.2 tokens), suggesting similar task complexity. In contrast, RefactorBench repos are larger (1149.2 files), and the issues are much shorter (14.6 tokens), creating a more challenging scenario with larger search spaces but minimal contextual information.

3.2 Additional experiment settings

To more systematically evaluate our benchmark contamination hypothesis, we design a set of complementary test conditions for the SWE-Bench-Verified subset. We consider three extensions: (1) Full-SWE-Bench, (2) SWE-Bench Extra, and (3) Outside-Repo Tasks, as detailed below.

Full-SWE-Bench is 200 instances randomly sampled tasks from the full SWE-Bench Jimenez et al. [2024] dataset, not included in the Verified subset. Since most existing work focuses exclusively on SWE-Bench-Verified, we hypothesize that if performance drops significantly on these other instances, it may indicate that models have been optimized specifically for the curated subset or have seen similar patterns during training, i.e., instance-specific memorization.

SWE-Bench Extra includes the recent issues that were mostly created after the original SWE-Bench dataset cutoff date sourced from SWE-Bench repositories as a complementary experiment. This includes 217 tasks from all repositories. These fresh tasks from the same repositories as SWE-Bench provide a critical test: if models perform significantly worse on recent issues from the same repos, where they excel on SWE-Bench tasks, it would strongly suggest that their success stems from instance-specific memorization, rather than repository-bias memorization.

Outside-Repo Tasks are tasks from repositories outside the SWE-Bench dataset but with a high likelihood of training data inclusion. We collected 245 SWE-Bench style instances from 7 popular repos: jupyter/notebook, celery/celery, aio-libs/aiohttp, scipy/scipy, numpy/numpy, pytorch/pytorch, and pandas-dev/pandas. These repos are widely used, well-documented, and have been publicly available for years. If there is strong performance on these extra-repo tasks, this would counter the possibility that models memorize the instances on SWE-Bench Verified. However, if models perform significantly better on SWE-Bench tasks than on these equally-exposed repo tasks, this would indicate that models are biased toward tasks sourced from repositories included in SWE-Bench, rather than developing general coding capabilities.

3.3 Metrics

In this section, we discuss metrics used in our experiments.

Accuracy (Acc.) The accuracy is computed as the percentage of predictions that exactly match the GT patch out of the total number of samples in that category, i.e., Accuracy (%) = $\frac{\text{Number of exact matched instances}}{\text{Total number of instances}} \times 100$. An exact match is recorded when the model-generated file path is identical to at least one of the reference GT file path(s).

Filtered accuracy (F-Acc.) To address concerns that models might succeed through superficial pattern matching rather than genuine understanding, we introduce filtered accuracy as a control metric. This measures performance exclusively on instances where problem descriptions do not explicitly contain file paths or import statements, identified through simple heuristics. By removing cases where path-like strings appear in issue descriptions, filtered accuracy isolates performance that depends purely on repository knowledge and problem-solving reasoning rather than surface-level string matching. This metric directly addresses our research question of whether models demonstrate genuine coding understanding or rely on memorized patterns.

5-gram Consecutive Overlap (5-gram Overlap.) To quantify the degree of verbatim memorization in model outputs, we compute the 5-gram overlap ratio, which measures the proportion of predicted 5-grams (consecutive sequences of 5 tokens) that exactly match those found in the GT code. N-gram Overlap (%) = $\frac{\text{Number of matched n-grams}}{\text{Total generated n-grams}} \times 100$. This metric is calculated as the ratio of matching 5-grams to the total number of 5-grams in the prediction, with frequency-aware matching that respects the occurrence count of each n-gram in the ground truth to prevent inflation from repeated phrases. This metric captures the extent to which models regurgitate exact token sequences. When models show high overlap on some benchmarks but not others, this differential pattern can indicate memorization of specific training data rather than consistent coding ability.

Instance-Level Verbatim Match Percentage For the prefix completion task, an instance is marked as memorized or compromised if the model generates at least one code hunk that exactly matches the corresponding GT implementation. A high instance-level verbatim match percentage suggests that many test instances may be untrustworthy, as portions of the output are memorized by the model. We report the percentage of compromised instances across the entire benchmark.

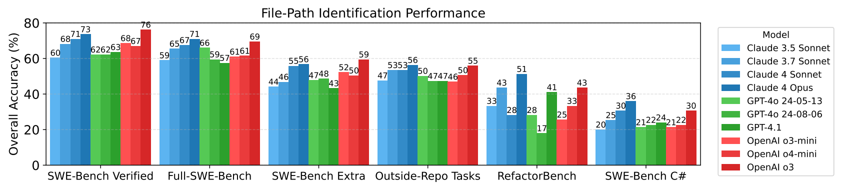

Figure 3: Comparison of model accuracy on the file path identification task across four benchmarks.

3.4 Models

We evaluated a diverse set of ten SOTA commercial LLMs from both OpenAI and Anthropic to measure their ability on the file path identification task. From OpenAI, we included snapshot variants of GPT-4o (gpt-4o-2024-08, and gpt-4o-2024-05), alongside GPT-4.1 (gpt-4.1-2025-04-14), reasoning-focused OpenAI o3 (o3-2025-04-16) and its lightweight counterpart o3-mini (o3-mini-2025-01-31), as well as o4-mini (o4-mini-2025-04-16), the compact successor OpenAI [2025]. From Anthropic, our suite comprised the Claude family: 3.5 Sonnet (June 20, 2024), 3.7 Sonnet (February 24, 2025), 4.0 Sonnet, and 4.0 Opus, both officially released on May 22, 2025. Model inference configurations are available in the appendix.

4 Results

4.1 File Path Identification Accuracy

Figure 3 presents the file path identification accuracy across all models and benchmarks, revealing a clear hierarchical pattern of performance. These results provide strong evidence for the two types of memorization, instance-specific and repository-bias, hypothesized in our introduction. We analyze the evidence for each pattern below.

4.1.1 Instance-Specific memorization

The first evidence pattern emerges from performance differences within the SWE-Bench repo ecosystem. Models consistently achieve their highest accuracy on the curated SWE-Bench-Verified subset (60-76%), with a noticeable decline on the broader Full-SWE-Bench set (57-71%) and a further drop on newly collected SWE-Repo tasks (50-68%).

This graduated performance decay suggests that models' knowledge is not uniform even across the same codebases. The superior performance on the "Verified" set, which is the most commonly used for evaluation and reporting, points toward instance-specific memorization, where models have been disproportionately exposed to or optimized for these specific, popular benchmark problems.

4.1.2 Repository-Bias memorization

The second, more dramatic pattern appears when comparing performance between SWE-Bench and external benchmarks, including RefactorBench and Outside-Repo Tasks. Tasks from both settings were sourced from popular repositories that tend to be well-represented in training data. If models developed a general capability for popular Python projects, we would expect strong performance here. Instead, accuracy is uniformly lower than that in SWE-Bench Verified.

This stark contrast between high performance on SWE-Bench repositories and poor performance on equally popular repositories reveals that model success depends on more than just repository popularity. It thus supports our hypothesis of repository-bias memorization, where models have overfit to the specific architectural patterns and problem distributions of the twelve repositories in the SWE-Bench collection, failing to develop transferable skills.

4.1.3 Model-Specific Contamination Patterns

The results reveal differences in contamination patterns across model families/vendors:

OpenAI chat Models: OpenAI's chat models show inconsistent performance patterns across benchmarks, with no stable ranking order. The performance ranking of these models varies significantly between different benchmark types.

OpenAI reasoning Models: While o3 demonstrates higher accuracy than o3-mini and o4-mini due to its enhanced capabilities or higher degree of overfitting, the performance patterns of o3-mini and o4-mini follow similar and inconsistent trends to the chat models.

Anthropic Models: Claude models exhibit more consistent performance patterns, with newer model generations showing higher path prediction accuracy. Overall, Sonnet 4 and Opus 4 maintain relatively higher performance across most benchmarks compared to other Claude models.

Cross-Vendor Consistency: A key finding from our evaluation of ten different LLMs is a remarkable consistency of these patterns across different model families and vendors. The fact that all models, regardless of vendor, exhibit the same performance hierarchy (SWE-bench Verified > Full-SWE-Bench > SWE-Repo Tasks > SWE-bench C#, RefactorBench, and Outside-Repo Tasks) indicates **the performance disparities reflect *systematic exposure patterns in training data* rather than issues isolated to a vendor**.

4.1.4 Implications for Benchmark Validity

The results provide strong empirical evidence for the two forms of memorization hypothesized in our introduction. The performance degradation within the SWE-Bench style benchmarks points to instance-specific memorization, while the more significant performance degradation on external repositories indicates repository-bias memorization. The presence of these two types of memorization suggests that high scores on SWE-Bench are not purely indicative of generalizable coding abilities for these models. Instead, they are likely inflated by these confounding factors. This shows that models possess specialized knowledge of specific tasks in SWE-Bench Verified and repository patterns, rather than or possibly in addition to transferable software engineering skills.

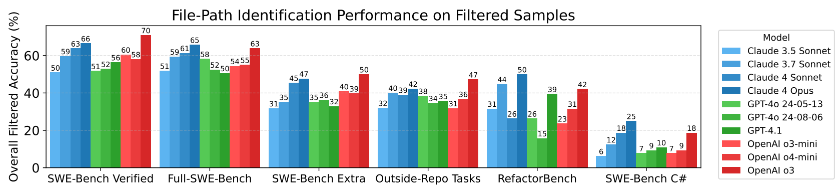

Figure 4: Comparison of models' filtered accuracy on the file path identification task across benchmarks.

4.2 File Path Identification Accuracy on Filtered Instances

To more rigorously test our contamination hypothesis, this section examines model performance only on instances where the issue description provides no explicit file paths or import statements. The results, presented in Figure 4, remove superficial string matching as a confounding variable and reveal that the underlying memorization patterns persist.

First, models still achieve the highest accuracy on SWE-Bench Verified instances, compared to Full SWE-Bench and SWE-Repo tasks, showing a pattern of instance-specific memorization inside SWE-Bench repositories. Second, the sharp performance drop on external repos also persists, providing evidence for repository-bias memorization.

The persistence of these patterns in the filtered data is our most critical finding. It shows that the models' ability to identify correct files is not due to simple text matching but is based on memorized knowledge, which includes instance-specific patterns and repository-biased patterns. This suggests that current benchmark scores are inflated by these memorization effects and don't purely reflect a model's transferable software engineering skills.

Figure 5: Instance-level verbatim match percentage across models on SWE-Bench Verified.

Model	Match (%)
Claude 3.5 Sonnet	12.1
Claude 3.7 Sonnet	12.3
Claude 4 Sonnet	21.4
Claude 4 Opus	31.6
GPT-4.1	17.4
GPT-4o 24-05-13	18.4
GPT-4o 24-08-16	18.2
OpenAI o3-mini	11.7

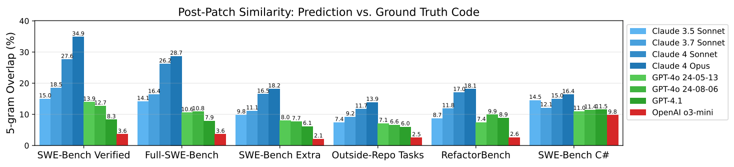

Figure 6: Comparison of models' 5-gram overlap ratio against the ground truth code snippet on the function reproduction task across benchmarks.

4.3 Function Reproduction Result

We analyze models' performance on the function reproduction task in this subsection. As shown in Figure 6, models achieve up to 34.9% 5-gram overlap ratio reproducing ground truth functions from SWE-Bench Verified compared to up to 18.1% on instances outside the SWE-Bench ecosystem[1]. With limited context provided, models lack sufficient information to systematically derive correct implementations. High consecutive 5-gram overlap under these constraints requires reproducing exact textual sequences, indicating reliance on memorized training data rather than reasoning.

Instance-Specific Memorization : Performance reveals a clear hierarchy among benchmarks: SWE Verified (34.9%) > SWE Full (28.7%) > SWE Extra (18.2%). Critically, SWE Extra performance drops to the same level as external benchmarks, despite originating from identical repositories as Verified. This pattern indicates that models have memorized these specific instances from SWE-Bench Full and Verified, rather than developing general familiarity with repositories.

4.4 Direct Memorization on SWE-Bench Verified

Table 5 presents the results of our prefix completion experiment across eight SoTA language models, showing substantial evidence of training data contamination in SWE-Bench Verified, with the instance-level verbatim match percentage ranging from 11.7% to 31.6% across evaluated models.

The Claude family exhibits a monotonic increase in memorization rates corresponding to model generation, rising from 12.1% to 31.6%. This progression suggests a strong correlation between model capacity and memorization. Notably, Claude 4 Opus demonstrates verbatim generation for nearly a third of all test instances. In contrast, the GPT family displays relatively stable memorization rates (17.4%-18.4%), indicating potential differences in training data curation or architectural choices that affect memorization. OpenAI o3-mini exhibits the lowest repro rate at 11.7%.

These results have significant implications for benchmark validity. The ability of models to reproduce exact code sequences when provided merely with contextual prefixes, without any problem description or bug identification, strongly suggests that performance on these instances reflects memorization rather than algorithmic reasoning or program comprehension. This finding strengthens our hypothesis that a substantial portion of reported performance gains on SWE-Bench may be attributed to memorization rather than genuine advances in programming capabilities.

5 Related Work

5.1 Coding Benchmarks and Agents

Various coding benchmarks have been proposed to evaluate LLMs and LLM agents, including SWE-Bench, BigCodeBench, SWE-Gym, and EvoEval Jimenez et al. [2024], Zhuo et al. [2024], Xia et al. [2024]. Among them, SWE-Bench has emerged as the most popular evaluation method for LLMs' coding abilities [Jimenez et al., 2024]. The benchmark consists of real GitHub issues, providing a seemingly realistic evaluation setting. However, the public nature of GitHub data raises concerns about potential exposure during training. Recent agent systems have shown dramatic improvements

[1]Due to limited resources, we only conduct experiments for `o3-mini` for the OpenAI reasoning models.

on SWE-Bench [Yang et al., 2024, Wang et al., 2024], but the extent to which these improvements reflect genuine problem-solving versus pattern matching remains unclear.

5.2 Benchmark Contamination

The issue of benchmark contamination, where models are inadvertently trained on eval data, is a well-documented problem that can lead to inflated performance metrics that don't reflect genuine understanding Zhou et al. [2023]. To combat this in the coding domain, prior work has developed two primary strategies: task mutation and metric-based probing.

Task mutation approaches aim to detect memorization by altering existing benchmarks. For example, EvoEval Xia et al. [2024] uses LLMs to create semantic-altering transformations of HumanEval Chen et al. [2021] problems. Similarly, other methods Chen et al. [2025] and TaskEval Tambon et al. [2025] create variations of a task and measure the performance discrepancy. However, these mutations only work for standalone tasks and do not work on SWE-Bench-style tasks that contain repository-level context. **Metric-based probing approaches**, in contrast, analyze model outputs without changing the task itself. These techniques Li [2023], Ramos et al. [2025] include using perplexity or other statistical measures like Negative Log Likelihood (NLL) and n-gram accuracy to infer if the model has likely seen a specific bug or solution before. However, NLL cannot be applied to commercial models since their hidden states are not accessible. N-gram similarity, while applicable, does not cover all phases of problem-solving for SWE-Bench-style tasks.

Our work introduces a novel cross-benchmark analysis framework that addresses these limitations by comparing models across related benchmarks as memorization proxies. Rather than requiring knowledge of training data exposure, we systematically compare model performance across benchmark variants and external evaluations to detect suspicious performance patterns. This provides a systematic, reusable framework for detecting memorization in any coding benchmark without requiring access to training data or model internals.

6 Limitations

We acknowledge two primary limitations. First, our black-box evaluation of commercial LLMs lacks access to training data or internal states, preventing definitive proof of memorization. Our conclusions rely on circumstantial evidence from performance disparities across diagnostic tasks. While we design three complementary tasks to detect memorization symptoms, we cannot directly observe the underlying mechanisms. Second, our cross-benchmark comparison approach could be influenced by inherent task difficulty variations beyond memorization effects. Although we observe similar repository size and issue description length across benchmarks, and find consistent patterns across ten models from different vendors, task-specific factors beyond our metrics might contribute to performance differences.

7 Conclusion

This work presents evidence that current evaluations of LLM coding abilities may be compromised by benchmark memorization. Our systematic evaluation across ten models reveals two distinct memorization patterns: instance-specific, evidenced by gradual performance decay within the SWE-Bench ecosystem, and repository-bias memorization, demonstrated by substantial performance drops on external repos (up to 47 percentage points for file-path identification) despite their popularity. The consistency of these patterns across different model families indicates that models were systematically exposed to SWE-Bench Verified data during training.

These findings suggest that the improvements in SWE-Bench Verified performance may partially reflect benchmark-specific optimization rather than genuine advances in coding capabilities. The field urgently needs more robust evaluation frameworks with temporal controls to prevent training data contamination, cross-repository validation to test generalization of these models beyond familiar codebases, and systematic cross-benchmark analysis to distinguish between memorization and transferable skills. Our work highlights the importance of developing contamination-resistant benchmarks and more sophisticated evaluation methodologies to ensure that reported progress reflects genuine advances in software engineering capabilities rather than overfitting on dataset-specific artifacts.

References

Anthropic. Introducing claude 4, May 2025. URL `https://www.anthropic.com/news/claude-4`. Blog post.

Samuel R. Bowman and George E. Dahl. What will it take to fix benchmarking in natural language understanding?, 2021. URL `https://arxiv.org/abs/2104.02145`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`.

Wentao Chen, Lizhe Zhang, Li Zhong, Letian Peng, Zilong Wang, and Jingbo Shang. Memorize or generalize? evaluating llm code generation with evolved questions, 2025. URL `https://arxiv.org/abs/2503.02296`.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. URL `https://arxiv.org/abs/2204.02311`.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code, 2025. URL `https://arxiv.org/abs/2503.07832`.

Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. 1(FSE), July 2024. doi: 10.1145/3660773. URL `https://doi.org/10.1145/3660773`.

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. URL `https://openreview.net/forum?id=VTF8yNQM66`.

Yucheng Li. Estimating contamination via perplexity: Quantifying memorisation in language model evaluation, 2023. URL `https://arxiv.org/abs/2309.10677`.

OpenAI. Introducing swe-bench verified. `https://openai.com/index/introducing-swe-bench-verified/`, 2024. URL `https://openai.com/index/introducing-swe-bench-verified/`. Published August 13, 2024; updated February 24, 2025.

OpenAI. Models · openai api, 2025. URL `https://platform.openai.com/docs/models`. API documentation page.

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. URL `https://arxiv.org/abs/2412.21139`.

Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. Are large language models memorizing bug benchmarks?, 2025. URL `https://arxiv.org/abs/2411.13323`.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, Ali Safaya, Ali Tazarv, Alice Xiang, Alicia Parrish, Allen Nie, Aman Hussain, Amanda Askell, Amanda Dsouza, Ambrose Slone, Ameet Rahane, Anantharaman S. Iyer, Anders Andreassen, Andrea Madotto, Andrea Santilli, Andreas Stuhlmüller, Andrew Dai, Andrew La, Andrew Lampinen, Andy Zou, Angela Jiang, Angelica Chen, Anh Vuong, Animesh Gupta, Anna Gottardi, Antonio Norelli, Anu Venkatesh, Arash Gholamidavoodi, Arfa Tabassum, Arul Menezes, Arun Kirubarajan, Asher Mullokandov, Ashish Sabharwal, Austin Herrick, Avia Efrat, Aykut Erdem, Ayla Karakaş, B. Ryan Roberts, Bao Sheng Loe, Barret Zoph, Bartłomiej Bojanowski, Batuhan Özyurt, Behnam Hedayatnia, Behnam Neyshabur, Benjamin Inden, Benno Stein, Berk Ekmekci, Bill Yuchen Lin, Blake Howald, Bryan Orinion, Cameron Diao, Cameron Dour, Catherine Stinson, Cedrick Argueta, César Ferri Ramírez, Chandan Singh, Charles Rathkopf, Chenlin Meng, Chitta Baral, Chiyu Wu, Chris Callison-Burch, Chris Waites, Christian Voigt, Christopher D. Manning, Christopher Potts, Cindy Ramirez, Clara E. Rivera, Clemencia Siro, Colin Raffel, Courtney Ashcraft, Cristina Garbacea, Damien Sileo, Dan Garrette, Dan Hendrycks, Dan Kilman, Dan Roth, Daniel Freeman, Daniel Khashabi, Daniel Levy, Daniel Moseguí González, Danielle Perszyk, Danny Hernandez, Danqi Chen, Daphne Ippolito, Dar Gilboa, David Dohan, David Drakard, David Jurgens, Debajyoti Datta, Deep Ganguli, Denis Emelin, Denis Kleyko, Deniz Yuret, Derek Chen, Derek Tam, Dieuwke Hupkes, Diganta Misra, Dilyar Buzan, Dimitri Coelho Mollo, Diyi Yang, Dong-Ho Lee, Dylan Schrader, Ekaterina Shutova, Ekin Dogus Cubuk, Elad Segal, Eleanor Hagerman, Elizabeth Barnes, Elizabeth Donoway, Ellie Pavlick, Emanuele Rodola, Emma Lam, Eric Chu, Eric Tang, Erkut Erdem, Ernie Chang, Ethan A. Chi, Ethan Dyer, Ethan Jerzak, Ethan Kim, Eunice Engefu Manyasi, Evgenii Zheltonozhskii, Fanyue Xia, Fatemeh Siar, Fernando Martínez-Plumed, Francesca Happé, Francois Chollet, Frieda Rong, Gaurav Mishra, Genta Indra Winata, Gerard de Melo, Germán Kruszewski, Giambattista Parascandolo, Giorgio Mariani, Gloria Wang, Gonzalo Jaimovitch-López, Gregor Betz, Guy Gur-Ari, Hana Galijasevic, Hannah Kim, Hannah Rashkin, Hannaneh Hajishirzi, Harsh Mehta, Hayden Bogar, Henry Shevlin, Hinrich Schütze, Hiromu Yakura, Hongming Zhang, Hugh Mee Wong, Ian Ng, Isaac Noble, Jaap Jumelet, Jack Geissinger, Jackson Kernion, Jacob Hilton, Jaehoon Lee, Jaime Fernández Fisac, James B. Simon, James Koppel, James Zheng, James Zou, Jan Kocoń, Jana Thompson, Janelle Wingfield, Jared Kaplan, Jarema Radom, Jascha Sohl-Dickstein, Jason Phang, Jason Wei, Jason Yosinski, Jekaterina Novikova, Jelle Bosscher, Jennifer Marsh, Jeremy Kim, Jeroen Taal, Jesse Engel, Jesujoba Alabi, Jiacheng Xu, Jiaming Song, Jillian Tang, Joan Waweru, John Burden, John Miller, John U. Balis, Jonathan Batchelder, Jonathan Berant, Jörg Frohberg, Jos Rozen, Jose Hernandez-Orallo, Joseph Boudeman, Joseph Guerr, Joseph Jones, Joshua B. Tenenbaum, Joshua S. Rule, Joyce Chua, Kamil Kanclerz, Karen Livescu, Karl Krauth, Karthik Gopalakrishnan, Katerina Ignatyeva, Katja Markert, Kaustubh D. Dhole, Kevin Gimpel, Kevin Omondi, Kory Mathewson, Kristen Chiafullo, Ksenia Shkaruta, Kumar Shridhar, Kyle McDonell, Kyle Richardson, Laria Reynolds, Leo Gao, Li Zhang, Liam Dugan, Lianhui Qin, Lidia Contreras-Ochando, Louis-Philippe Morency, Luca Moschella, Lucas Lam, Lucy Noble, Ludwig Schmidt, Luheng He, Luis Oliveros Colón, Luke Metz, Lütfi Kerem Şenel, Maarten Bosma, Maarten Sap, Maartje ter Hoeve, Maheen Farooqi, Manaal Faruqui, Mantas Mazeika, Marco Baturan, Marco Marelli, Marco Maru, Maria Jose Ramírez Quintana, Marie Tolkiehn, Mario Giulianelli, Martha Lewis, Martin Potthast, Matthew L. Leavitt, Matthias Hagen, Mátyás Schubert, Medina Orduna Baitemirova, Melody Arnaud, Melvin McElrath, Michael A. Yee, Michael Cohen, Michael Gu, Michael Ivanitskiy, Michael Starritt, Michael Strube, Michal Swedrowski, Michele Bevilacqua, Michihiro Yasunaga, Mihir Kale, Mike Cain, Mimee Xu, Mirac Suzgun, Mitch Walker, Mo Tiwari, Mohit Bansal, Moin Aminnaseri, Mor Geva, Mozhdeh Gheini, Mukund Varma T, Nanyun Peng, Nathan A. Chi, Nayeon Lee, Neta Gur-Ari Krakover, Nicholas Cameron, Nicholas Roberts, Nick Doiron, Nicole Martinez, Nikita Nangia, Niklas Deckers, Niklas Muennighoff, Nitish Shirish Keskar, Niveditha S. Iyer, Noah Constant, Noah Fiedel, Nuan Wen, Oliver Zhang, Omar Agha, Omar Elbaghdadi, Omer Levy, Owain Evans, Pablo Antonio Moreno Casares, Parth Doshi, Pascale Fung, Paul Pu Liang, Paul Vicol, Pegah Alipoormolabashi, Peiyuan Liao, Percy Liang, Peter Chang, Peter Eckersley, Phu Mon Htut, Pinyu Hwang, Piotr Miłkowski, Piyush Patil, Pouya Pezeshkpour, Priti Oli, Qiaozhu Mei, Qing Lyu, Qinlang Chen, Rabin Banjade, Rachel Etta Rudolph, Raefer Gabriel, Rahel Habacker, Ramon Risco, Raphaël Millière, Rhythm Garg, Richard Barnes, Rif A. Saurous, Riku Arakawa, Robbe Raymaekers, Robert Frank, Rohan Sikand, Roman Novak, Roman Sitelew, Ronan LeBras, Rosanne Liu, Rowan Jacobs, Rui Zhang, Ruslan Salakhutdinov, Ryan Chi, Ryan Lee, Ryan Stovall, Ryan Teehan, Rylan Yang, Sahib Singh, Saif M. Mohammad, Sajant Anand, Sam Dillavou, Sam Shleifer, Sam Wiseman, Samuel Gruetter, Samuel R. Bowman, Samuel S. Schoenholz, Sanghyun Han, Sanjeev Kwatra, Sarah A. Rous, Sarik Ghazarian, Sayan Ghosh, Sean Casey, Sebastian Bischoff, Sebastian Gehrmann, Sebastian Schuster, Sepideh Sadeghi, Shadi Hamdan, Sharon Zhou, Shashank Srivastava, Sherry Shi, Shikhar Singh, Shima Asaadi, Shixiang Shane Gu, Shubh Pachchigar, Shubham Toshniwal, Shyam Upadhyay, Shyamolima, Debnath, Siamak Shakeri, Simon Thormeyer, Simone Melzi, Siva Reddy, Sneha Priscilla Makini, Soo-Hwan Lee, Spencer Torene, Sriharsha Hatwar, Stanislas Dehaene, Stefan Divic, Stefano Ermon, Stella Biderman, Stephanie Lin, Stephen Prasad, Steven T. Piantadosi, Stuart M. Shieber, Summer Misherghi, Svetlana Kiritchenko, Swaroop Mishra, Tal Linzen, Tal Schuster, Tao Li, Tao Yu, Tariq Ali, Tatsu Hashimoto, Te-Lin Wu, Théo Desbordes, Theodore Rothschild, Thomas Phan, Tianle Wang, Tiberius Nkinyili, Timo Schick, Timofei Kornev, Titus Tunduny, Tobias Gerstenberg, Trenton Chang, Trishala Neeraj, Tushar Khot, Tyler Shultz, Uri Shaham, Vedant Misra, Vera Demberg, Victoria Nyamai, Vikas Raunak, Vinay Ramasesh, Vinay Uday Prabhu, Vishakh Padmakumar, Vivek Srikumar, William Fedus, William Saunders, William Zhang, Wout Vossen, Xiang Ren, Xiaoyu Tong, Xinran Zhao, Xinyi Wu, Xudong Shen, Yadollah Yaghoobzadeh, Yair Lakretz, Yangqiu Song, Yasaman Bahri, Yejin Choi, Yichi Yang, Yiding Hao, Yifu Chen, Yonatan Belinkov, Yu Hou, Yufang Hou, Yuntao Bai, Zachary Seid, Zhuoye Zhao, Zijian Wang, Zijie J. Wang, Zirui Wang, and Ziyi Wu. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2023. URL `https://arxiv.org/abs/2206.04615`.

Florian Tambon, Amin Nikanjam, Cyrine Zid, Foutse Khomh, and Giuliano Antoniol. Taskeval: Assessing difficulty of code generation tasks for large language models, 2025. URL `https://arxiv.org/abs/2407.21227`.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL `https://arxiv.org/abs/2302.13971`.

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. Opendevin: An open platform for ai software developers as generalist agents, 2024. URL `https://arxiv.org/abs/2407.16741`.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint*, 2024.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. Don't make your llm an evaluation benchmark cheater, 2023. URL `https://arxiv.org/abs/2311.01964`.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Supplementary: SWE-Bench C# Details

We show the repository distribution of SWE-Bench C# in Figure 7. This benchmark is useful for tuning agents problem-solving approach for bugfixing, feature addition, and refactoring in the C# and .NET ecosystem.

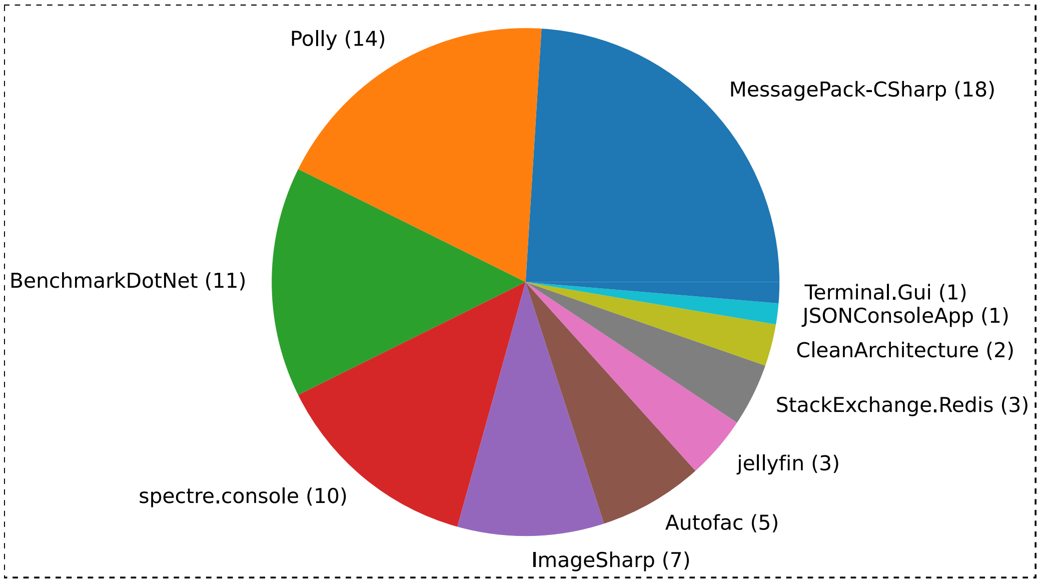

Figure 7: Distribution of repositories in SWE-Bench C#.

B Supplementary: Prompt Templates and Examples

B.1 File-Path Identification Prompt Template

We present the prompt template for the File-Path Identification task in Figure 8. The prompt begins with the repository name, followed by the issue description, which serves as the problem statement for SWE-Bench Verified tasks, and concludes with instructions for the model to reason through the problem and output a single predicted buggy file path.

B.2 SWE-Bench Verified Task Example

Figure 9 is a reduced example for SWE-Bench Verified instance with ID: scikit-learn__scikit-learn-10297. In practice, problem statements similar to this will be provided to LLMs in the file-path identification task, and the function reproduction task as a part of the prompt.

B.3 SWE-Bench C# Example

Figure 10 shows the problem statement for instance: ardalis__cleanarchitecture-530 in SWE-Bench C#.

B.4 Function Reproduction Task Prompt Template

We present the prompt template for the Function Reproduction task in Figure 11. The prompt begins with the repository name, followed by hints indicating which functions need to be implemented, and only the names and the file paths are included in the hints. It then provides the complete file content, where the buggy functions are removed and replaced by comments marking the index of each missing function. Finally, the prompt instructs the model to reason through the task and output the reconstructed content for all buggy functions identified in the provided file.

Complete File Path Prompt Implementation

```
def get_file_path_prompt(entry):
   file_prompt = '''You will be provided with an issue statement explaining a problem to
resolve within a codebase. The code base is: {0}.
  <issue>
  {1}
  </issue>
  You must always include one discussion and one response as part of your response.
  Make sure you do not have multiple discussion/response tags.
  Please make sure your output precisely matches the following format.
  DISCUSSION
  Discuss here with yourself about how you came up with this response.
  RESPONSE
  ```
  response(s) to the question asked.
  ```
  Now answer the following question:
  Given the issue description and project, provide a file-path of the .py
file containing the issue relative to the root.'''.format(entry['repo'],
entry['problem_statement'])
   model_input = [{
     'role': 'user',
     'content': file_prompt
   }]
   return model_input
```

Figure 8: The complete implementation of the file path identification prompt used in our experiments.

RidgeClassifierCV store_cv_values Issue

Error: `TypeError: __init__() got an unexpected keyword argument 'store_cv_values'`
Repro:

```
from sklearn import linear_model
import numpy as np

x = np.random.randn(100, 30)
y = np.random.normal(size=100)

model = linear_model.RidgeClassifierCV(
     alphas=np.arange(0.1, 1000, 0.1),
     normalize=True,
     store_cv_values=True
).fit(x, y)
```

Note: `store_cv_values` is not a valid parameter, yet some attributes depend on it.

Figure 9: Minimal reproduction of RidgeClassifierCV parameter issue

C Supplementary: Additional Experiment Results

C.1 Comparison of 5-Gram Overlap for Buggy V.S. Ground Truth Code Snippet

We also report the difference of 5 gram overlap

$$\Delta_5 = \text{overlap}_5(\hat{f}, f_{\text{GT}}) - \text{overlap}_5(\hat{f}, f_{\text{buggy}}),$$

Issue: Request to Create New ToDoItems, instance: [ardalis__cleanarchitecture-530]

```
Please show how you would create new ToDoItems
<!--  Do Not Delete This! feature_request_template -->
<!-- Please search existing issues to avoid creating duplicates. -->

<!-- Describe the feature you'd like. -->

In this template, you aggregate ToDoItems within the ProjectAggregate.
However, you also stop short of showing how you would, via API,
author new ToDoItem instances. I think it would be beneficial to
show how you would create these new child items of a project.

I think it would be wonderful to see:
- How you would define this in an ApiEndpoint.
- How you would incorporate this within the ProjectAggregate.
- How you would codify a Many-to-Many relationship as opposed to
    the One-To-Many relationship between Project and ToDoItem.
- Is this implied by the Project <-> Contributor relationship?
- What if the Contributor to a Project had a title within that Project?
    Senior Contributor vs Junior Contributor? If that were the case,
    what ApiEndpoint would the management of that be within your example
    domain?

Thanks for such a great template!
```

Figure 10: Example issue description requesting creation of new `ToDoItem` instances

where overlap$_5(x, y)$ is the 5 gram overlap of x and y. Positive values indicate the model output more closely matches the patched implementation; negative values indicate closer similarity to the buggy code.

Figure 12 summarizes Δ_5 across benchmarks and model families. Three broad patterns emerge.

(1) Provider-level asymmetry on legacy benchmarks. Anthropic's Claude models show consistently positive differentials on SWE-BENCH VERIFIED, FULL-SWE-BENCH, and OUTSIDE-REPO TASKS (roughly +2 to -6 percentage points), suggesting that these models retain verbatim shards of the patched implementations. OpenAI models yield smaller positives (generally less than +2pp) over the same sets, indicating weaker memorization of those specific fixes.

(2) Freshness stress test. All models, except Claude 3.5 Sonnet and GPT 4.1, produce *negative* Δ_5 on SWE-BENCH EXTRA. Because this split is mined largely from post-cutoff commits, the patches are unlikely to be in training data. The systematic shift toward the buggy reference therefore supports our contamination-mitigation claim: when the patched code is unseen, models do not memorize it from context.

(3) Language / ecosystem skew. On the SWE C# subset, OpenAI's GPT-4* family exhibits the largest positive differentials (~+3 to 5pp), exceeding Claude. This may suggest heavier exposure to patched C# / .NET code in OpenAI training data (or lighter deduplication of that ecosystem) compared to Anthropic.

Taken together, the function reproduction experiment provides a direct, contrastive probe of code memorization: when historical (patched) code is in a model's training mix, we observe measurable positive Δ_5; when the code post-dates the model (SWE-BENCH EXTRA), the signal disappears or reverses. Additionally, the high tendency towards the fixed version of code on SWE-Bench Verified and SWE-Bench C# indicates higher possibility of benchmark memorization on these two settings. These results underscore the necessity of freshness-controlled benchmarks when interpreting coding model performance and memorization risk.

Function Reproduction Task Prompt Template

~~~
You are provided with a code repository and an issue description. Your task
is to implement the complete function bodies for the marked RESPONSE comments.
These functions are necessary to resolve the issue described in the problem
statement.
Repository: scikit-learn/scikit-learn
<issue>
Issue description here
</issue>
...
HINT: Fixes in _tags.py: modification in get_tags
=== File: sklearn/utils/_tags.py ===
<file>
...
#TODO: RESPONSE 1:
...
</file>
...
You must provide complete implementations for each RESPONSE marker. Each
function should:
1. Be properly indented to match its position in the file
2. Include the complete function body including the function signature (def
line)
3. Address the issue described above
RESPONSE INSTRUCTIONS:
RESPONSE 1: Complete implementation of get_tags in sklearn/utils/_tags.py
IMPORTANT:
- Provide the COMPLETE function implementation, including the 'def' line and
all decorators if any
- Maintain correct Python indentation
- Each RESPONSE should contain the entire function from decorators (if any)
through the complete function body
- Number your responses to match the RESPONSE markers in the code
- Wrap each response in triple backticks (```)
Please think through the issue first, then provide your implementations.
DISCUSSION:
Analyze the issue and plan your implementations here
RESPONSE 1:
```python
# Your implementation here
```
~~~

Figure 11: The prompt template used to test models' ability to reproduce ground truth function without function specification.

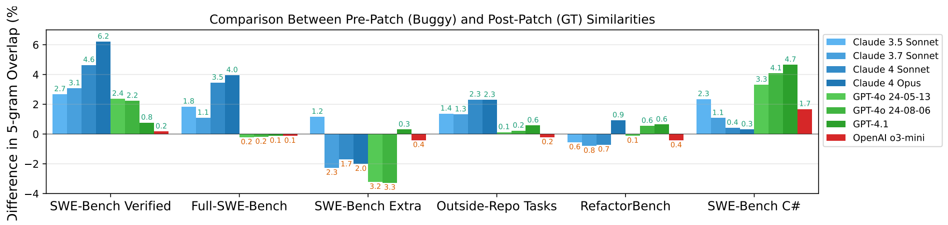

Figure 12: Difference in models' 5-gram overlap ratio when comparing against the ground truth code snippet versus the buggy code snippet in the function reproduction task across benchmarks. A higher difference indicates that the model's solution is more similar to the ground truth.