Are Code LLMs as Intelligent as Leaderboards Suggest? An Empirical Study of Data Contamination Effects

Anonymous ACL submission

Abstract

Code large language models (Code LLMs) have achieved remarkable progress across software engineering tasks, yet their true generalization ability remains unclear due to data contamination—benchmark code leaking into training corpora. Unlike in NLP, contamination in Code LLMs is underexplored despite the prevalence of public benchmarks with canonical solutions, making them especially vulnerable to memorization. This paper presents a comprehensive study of contamination in open-source Code LLMs. We aim to (1) quantify benchmark contamination across existing models and (2) assess the effectiveness of mitigation techniques, including post-hoc detection via membership inference and dynamic benchmarking. Through systematic experiments on diverse models, tasks, and datasets, we find that (i) popular benchmarks are already contaminated, inflating reported performance; (ii) membership-inference attacks fail to reliably detect contamination, performing no better than random guessing; and (iii) dynamic benchmarking, while conceptually appealing, introduces practical challenges and potential biases in fair evaluation. Our findings call for contamination-aware evaluation protocols and the development of cleaner, more adaptive benchmarks. Our code and experimental data could be found at the anonymous link. [1]

1 Introduction

Code large language models (Code LLMs) have recently demonstrated remarkable progress in addressing a variety of challenging software engineering tasks, including code generation, code completion, and automatic code repair (Brown et al., 2020; Guo et al., 2024; Zheng et al., 2024; Luo et al., 2025; DeepSeek-AI et al., 2024; Yu et al., 2024a). These models are typically trained on massive corpora of both source code and natural language, and their performance is evaluated on public benchmarks such as HumanEval—a widely recognized gold standard for code generation (Chen et al., 2021; Liu et al., 2023; Zhuo et al., 2025).

Although the accuracy scores on these well-established leaderboards have improved dramatically in recent years, there remains a critical concern: *Are Code LLMs truly as intelligent as these leaderboard results suggest?* Because the training data of large Code LLMs is vast and opaque, it is often unclear whether public benchmarks have been included—intentionally or not—in their training corpora. This data contamination can cause overfitting to benchmark tasks, creating a misleading impression of model intelligence. Verifying contamination is especially challenging, as data privacy and proprietary constraints obscure access to training datasets.

Prior work has shown data contamination in NLP benchmarks, suggesting that public code benchmarks are likely vulnerable as well (Chen et al., 2025a; Magar and Schwartz, 2022; Liang et al., 2023; Aiyappa et al., 2023). To address potential contamination, several automated mitigation strategies have been proposed. These strategies include: (1) Post-hoc contamination detection (Duan et al., 2024; He et al., 2025; Shi et al., 2024), which originates from membership inference attacks and seeks to determine whether a particular data sample was included in a model's training set; and (2) Dynamic benchmarking (Li et al., 2024; Wang et al., 2024; Fan et al., 2024), which generates new and diverse benchmark samples at evaluation time by modifying existing data, thereby reducing the risk of contamination. However, the effectiveness of these methods—originally designed for smaller or NLP-focused models—remains unclear for Code LLMs due to their complex pre-training and fine-tuning pipelines and the distinct nature of code evaluation, which relies on execution rather than text matching.

[1] https://github.com/AnonymousHawk-code/CodeLLMDataContamination

To bridge these gaps, we conduct an empirical study on open-source Code LLMs to (1) quantify data contamination and (2) assess the effectiveness of mitigation strategies such as post-hoc detection and dynamic benchmarking. We address the following research questions (RQs):

RQ1 Contamination Status: Are current Code LLMs already contaminated with benchmark data, and if so, to what extent does this contamination occur?

RQ2 Effectiveness of Post-hoc Detection: How effective are existing post-hoc contamination detection methods in identifying contaminated samples within Code LLMs?

RQ3 Effectiveness of Dynamic Benchmarking: How well do dynamic benchmarking strategies mitigate the impact of data contamination when evaluating Code LLMs?

To answer the above research questions, we conduct a series of comprehensive experiments. For **RQ1**, we evaluate four Code LLMs and cut off the dataset in to two parts: one collected before the model's release and one after. We randomly sample an equal number of examples from each dataset with comparable complexity and measure the models' *pass@1* scores, allowing us to quantify memorization of pre-release benchmark data and its impact on generalization to new examples. For **RQ2**, we artificially contaminate four Code LLMs by fine-tuning them on benchmark data, and then assess the effectiveness of three state-of-the-art post-hoc detection methods in identifying contaminated samples. For **RQ3**, we examine two representative dynamic benchmarking methods, evaluating not only their ability to accurately reveal the true capabilities of Code LLMs in the presence of data contamination, but also their potential drawbacks, such as inadvertently impairing model performance or fairness in evaluation.

Our study reveals three key insights on Code LLM evaluation: (1) models often memorize pre-release benchmark data, inflating performance on static benchmarks and dropping on unseen tasks; (2) membership inference–based post-hoc detection is largely ineffective; and (3) dynamic benchmarking shows promise but is inconsistent and may add evaluation complexity. These findings underscore the need for more robust, dynamic evaluation frameworks. We summarize our contributions as follows:

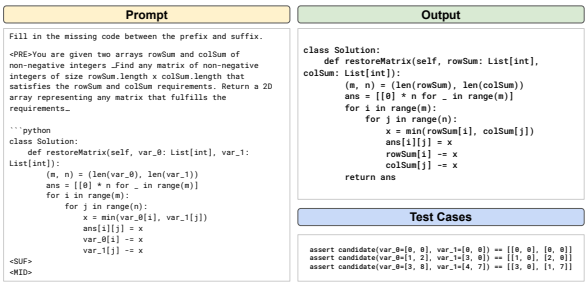

Figure 1: LLM Memorization Example

- We conduct the first empirical study to understand data contamination issues in current Code LLMs, and our results indicate that existing Code LLMs are already contaminated on popular benchmarks.
- We investigate the effectiveness of three well-established membership inference attack algorithms for detecting contamination in Code LLMs. Our findings show that their performance is comparable to random guessing.
- We evaluate the effectiveness of dynamic benchmarking methods in mitigating data contamination. The results suggest that dynamic benchmarking can reduce contamination to some extent; however, these methods are not foolproof, as they may introduce additional and unpredictable benchmark complexity.

2 Motivation Example

Figure 1 illustrates an example of this contamination. When evaluated under parameter standardization, the model outputs a solution using the original variable names, causing the test case to fail despite the correctness of the underlying algorithm. Parameter standardization reduces lexical bias by renaming all function parameters according to a uniform convention (e.g., var_0, var_1, ...) across both function bodies and test cases. The goal is to assess whether models can perform semantic reasoning when superficial lexical cues are removed. Yet, under this seemingly simple transformation, we observed a sharp performance drop. Models frequently reproduced memorized LeetCode solutions complete with the original parameter names even though the rest of the code and context had been normalized. As a result, their outputs failed standardized test cases, despite being logically correct in algorithmic structure.

This example underscores why contamination studies are essential. Without accounting for memorization, models may seem capable of solving complex tasks, when in reality they succeed only because benchmark problems or their lexical artifacts are leaked into training data. To the best of our knowledge, we are the first to systematically evaluate contamination across multiple software engineering tasks using diverse approaches (including temporal analysis and membership-inference attacks) and to compare these results against dynamic benchmarking strategies to assess their effectiveness.

3 Methodology

3.1 Code LLMs

To conduct our experiments, we employ a diverse set of open-source code LLMs spanning sizes, architectures, and training objectives. For RQ1 and RQ3, we use mid-sized instruction-tuned models (6–8B) to balance performance and inference cost: Llama-3.1-8B-Instruct, Qwen2.5-7B-Instruct, Qwen2.5-Coder-7B-Instruct, and DeepSeek-Coder-6.7B-Instruct. For RQ2, which involves training from scratch or fine-tuning on controlled datasets, we use smaller models due to compute constraints: Llama-3.2-1B, Qwen2.5-0.5B, Qwen2.5-Coder-0.5B, and DeepSeek-Coder-1.3B-Base.

3.2 Datasets

To comprehensively understand data contamination issues when applying large language models (LLMs) to software engineering tasks, we focus on two widely studied tasks: code generation (CG) and code infilling (CI). These tasks are representative of real-world software development scenarios and serve as core benchmarks for evaluating the practical utility and reliability of LLMs in software engineering applications. Code Generation involves producing source code from a natural language prompt, with the input being a textual description and the output a code snippet in the target language. Code Infilling focuses on filling in missing code between a given prefix and suffix; the input consists of the prefix and suffix, and the output is the intermediate code segment.

Code Generation. We use 3 datasets: HumanEval, MBPP-Sanitized, LeetCode. HumanEval is an open source benchmark consisting of 164 Python programming problems released by OpenAI. MBPP-Sanitized is a crowd-sourced dataset of 427 basic Python programming problems from Google. Leetcode is a set of 2869 Python programming problems collected from the LeetCode coding platform.

Code Infilling. To construct infilled versions of each dataset, we take every problem and iteratively remove one line from its solution. This produces n subproblems, each consisting of a prefix, a suffix, and the target line to be predicted.

Detailed description of each dataset used can be found in Appendix A.

3.3 Experimental Process

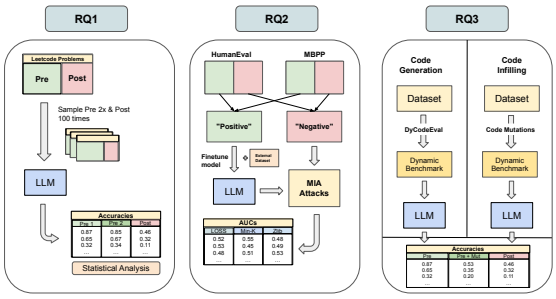

Figure 2: Summary of each RQ experiment

RQ1 Process. To investigate whether current code LLMs show signs of memorization on software engineering tasks, we compare their performance on LeetCode problems released before and after the models' training cutoff. Cutoff dates were taken from the official HuggingFace website as the most recent update date for each model. To ensure consistency across multiple models, we standardized the split using the latest cutoff. Problems released after this point could not have appeared in training and therefore provide a cleaner measure of generalization. We establish a baseline by measuring accuracy on a sample of pre-cutoff problems, then compare this against performance on both pre- and post-cutoff sets, repeating the experiment 100 times to account for variance. We use a sample size of 40-50 problems for each problem set depending on availability and randomly sample with replacement. To ensure fairness, pre- and post-cutoff problems are matched in difficulty. We approximate complexity using three indicators: (i) prompt token length, (ii) solution line length, and (iii) Leetcode labeled difficulty ("Easy," "Medium," "Hard"). These

Approach	Temporal Analysis	Mutation Analysis	MIA LLM Analysis	Multiple SE Tasks	Controlled
LiveCodeBench (Jain et al., 2024)	✓	—	—	×	×
To the Cutoff (Roberts et al., 2024)	✓	—	—	×	✓
Concerned with... (Cao et al., 2024)	✓	✓	—	×	×
ReCode (Wang et al., 2022)	—	✓	—	×	—
Mutation Consistency Study (Li and Shin, 2024)	—	✓	—	×	—
Do MIA attacks work? (Duan et al., 2024)	—	—	✓	×	—
Ours	✓	✓	✓	✓	✓

Table 1: Differences between our study and the different data contamination approaches. Legend: ✓ = present; × = absent; — = not applicable

measures allow us both to control for complexity and to test consistency across a range of task difficulties.

To evaluate whether observed performance differences are statistically significant, we conduct an independent two-sample t-test on the Pass@1 scores from the pre- and post-cutoff problem sets. The Pass@1 metric is continuous, and the two groups (pre vs. post) are independent. The distributions of the Pass@1 scores approximate normality given repeated sampling (n = 40-50 per group) and the variances are compared; If unequal, we use Welch's correction. This setup allows appropriate interpretation of whether cutoff date significantly impacts model performance.

RQ2 Process. In this research question, we examine the effectiveness of post-hoc detection algorithms, specifically membership inference attacks (MIAs), in identifying data contamination in code LLMs. We aim to determine whether these post-hoc techniques can reliably identify whether a model has been exposed to specific data during training. To conduct this analysis, we construct two datasets, each comprising of half HumanEval and half MBPP problems. One dataset is designated as the positive set (simulating memorized data), and the other as the negative set (unseen data). We evaluate a range of models using three established membership inference attack (MIA) techniques: Loss-based, Min_K, and Zlib-based attacks. Each method aims to distinguish between samples the model has seen during training (members) and those it has not (non-members), based on differences in model behavior.

Loss-Based Attack: Based on the assumption that a model typically assigns lower loss values to samples it has seen during training, as compared to unseen examples. Let x be an input and y its true label or expected output. The loss-based score is computed using the model's loss function $L(f(x), y)$ where $f(x)$ is the model's output.

$$Score_{Loss}(x, y) = -L(f(x), y) \quad (1)$$

Min_K Attack: Computes the average log-probability over the K least likely tokens in a sample, aiming to capture weak points where the model's uncertainty would be greatest for unseen examples. Let x = $(x_1, x_2, ..., x_T)$ be a tokenized sequence. Let $l_i = logP(x_i|x_{<i})$ be the log-probability assigned by the model to token x_i

$$Score_{Min_K}(x) = \frac{1}{K} \sum_{i \in I_K} l_i \quad (2)$$

where $I_K \in 1...T$ is the set of the K tokens with the lowest log probabilities.

ZLib Attack: Compression-based method that uses file compression as a proxy for data familiarity. Let $C(s)$ denote the size in bytes of the compressed string s using the Zlibm compressor. The Zlib score is defined as

$$Score_{Zlib}(x) = \frac{1}{C(x)} \quad (3)$$

We first establish a baseline by running the attacks on models that have not been exposed to either dataset and computing their AUC (Area Under the Curve) scores. To simulate contamination, we then intentionally introduce the positive dataset into each model's training data and reevaluate performance. Specifically, we fine-tune existing models on a larger corpus that includes the positive set (Ex. Leetcode + positive dataset), using the HuggingFace Trainer class with a batch size of 32 and a learning rate of $2e^{-5}$ for 1 epoch. This setup enables us to assess how effectively each attack method can distinguish between memorized and non-memorized examples.

```
def filter_numbers(nums, threshold):
    result = []
    for num in nums:
        if num > threshold:
            result.append(num)
        else:
            result.append(-1)
    return result
```

Original Code

```
def filter_numbers(nums, threshold):
    result = []
    for num in nums:
        num = num + 0
        if num > threshold:
            result.append(num)
        else:
            result.append(-1)
    return result
```

Identity Operations (IO)

```
def filter_numbers(nums, threshold):
    result = []
    i = 0
    while i < len(nums):
        if num > threshold:
            result.append(num)
        else:
            result.append(-1)
        i += 1
    return result
```

For to While (FW)

```
def filter_numbers(nums, threshold):
    result = []
    for num in nums:
        if not (num > threshold):
            result.append(-1)
        else:
            result.append(num)
    return result
```

If-Else Switch (IE)

Figure 3: Code Mutation Examples

RQ3 Process. To evaluate the effectiveness of dynamic benchmarking, we will conduct an experiment comparing model performance on original versus mutated benchmark data across two software engineering tasks: code generation and code infilling. For code generation, where the input consists solely of natural language, we will apply dynamic prompt mutation using the diversification strategy employed by DyCodeEval (Chen et al., 2025b). For code infilling, we will mutate the code inputs provided to the model, focusing on two main categories: Code Formatting and Code Structure. Code formatting mutations are particularly valuable, as they assess a model's ability to solve a task without relying on superficial or memorized cues. The following code formatting mutation techniques will be applied:

Parameter Standardization: We rename all parameters in a sequential convention to a standardized format (e.g. $rowSum \rightarrow var_0$)

Identity Operations: We add an extra addition/multiplication identity operation to one of the variables.

Code structure mutations provide a way to test a model's ability to understand how information and logic flows through the code while preserving semantics. We will run the following code structural mutations:

For-While Transformation: We convert all the for loops in the given code to while loops.

If-Else Switch: We reverse the order of the if-else statements.

The set of applied mutations is illustrated in Figure 3. We focus on a set of commonly used mutations for evaluating LLM robustness and potential contamination (Wang et al., 2022; Li and Shin, 2024). Model performance under these mutations is measured using the Pass@1 metric.

3.4 Data Analysis

RQ1: To address this research question, we define Pass@1 drop as the decrease in accuracy from the baseline model to the pre- and post-cutoff versions. Our analysis focuses on comparing two key values: Pass@1 drop (baseline $\rightarrow$ pre-cutoff) and Pass@1 drop (baseline $\rightarrow$ post-cutoff). If these drops are statistically similar, it suggests that the model is reasoning rather than relying on memorized solutions. In contrast, a statistically significant difference between the two would indicate that the model has been leveraging memorized benchmark data.

RQ2: To interpret the results of this research question, we focus on the change in AUC (Area Under the Curve) between the baseline and the contaminated models. An increase in AUC indicates that the attack method is effective at distinguishing between seen and unseen data, suggesting successful detection of contamination. Conversely, if the AUC decreases or remains near 0.5, it implies that the method performs no better than random guessing. In such cases, we conclude that the attack fails to reliably detect contamination, and thus, lacks practical utility in this context.

RQ3: As we compare model performance on original benchmarking versus dynamic benchmarking, our goal is to determine whether accuracy declines under dynamic conditions. A performance drop would suggest that the model may have memorized benchmark data, and that altering the problems reduces its effectiveness. Consistently lower performance on dynamic benchmarks would therefore indicate a more accurate assessment of the model's reasoning abilities. Conversely, if the model performs similarly in both settings, it suggests that current dynamic benchmarking approaches may not sufficiently address the issue of data contamination.

3.5 Replication Package

All experiments were conducted on a server equipped with 8 NVIDIA RTX A6000 GPUs and

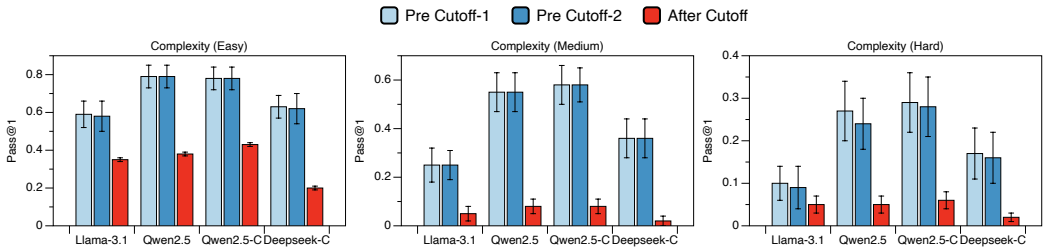

Figure 4: Code Generation Benchmarking Results Pre and Post Model Cutoff

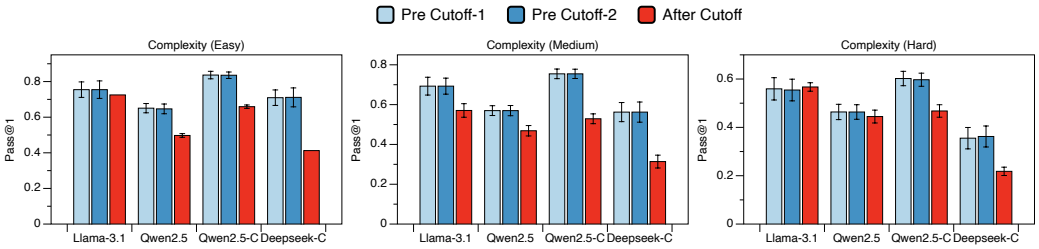

Figure 5: Code Infilling Benchmarking Results Pre and Post Model Cutoff

an AMD EPYC 9354 32-Core Processor. The implementation was carried out using the PyTorch, Transformers, and vLLM libraries. For code generation, inference was performed with a temperature of 0 and top-p sampling set to 0.95.

4 Results and Discussion

4.1 RQ1 Results

Main Results. Figures 4 and 5 show benchmark memorization results for code generation and infilling. For code generation, Pass@1 drops negligibly from baseline to pre-cutoff, indicating stable performance on pre-release samples. In contrast, post-cutoff scores fall sharply—50–60% for Easy and 80–90% for Hard problems—revealing significant contamination. Llama-3.1-8B-Instruct is an exception, showing more stable performance. These results suggest that high scores on challenging problems often reflect memorization rather than genuine problem-solving ability.

A similar trend appears in code infilling (Figure 5), where models achieve higher overall Pass@1 due to task simplicity. Still, post-cutoff scores drop noticeably across all models and datasets, showing that contamination affects even straightforward infilling tasks. These results highlight the need for evaluation protocols that account for temporal splits and benchmark freshness. Additional supporting results for benchmarking based on input token and solution length can be found in Appendix B.

Statistical Results. We then conducted independent two-sample t-tests on Pass@1 scores to assess the statistical significance of performance differences across datasets. The first test (pre-cutoff 1 vs pre-cutoff 2) yielded high p-values (> 0.05), showing no significant difference between the pre-cutoff datasets and implying comparability in model familiarity. In contrast, the second test (pre-cutoff 2 vs post-cutoff) produced very low p-values (< 0.05) and large t-values across all models and difficulty levels, confirming that the models' accuracy drops substantially on post-cutoff data. These results indicate that the decline reflects genuine challenges for Code LLMs on novel, uncontaminated data rather than random variation.

> Answers to **RQ1**: Our experiments show that Code LLMs have memorized parts of benchmark data available before their release, causing performance to drop notably on unseen but similar tasks.

4.2 RQ2 Results

Table 3 reports the AUC scores of various post-hoc detection methods—loss, min_k, and zlib—on different Code LLMs, with each model fine-tuned on a dataset that mixes contaminated samples with two other datasets. Across all models and detection methods, the AUC scores consistently hover around 0.5–0.57, only marginally above the random guessing baseline (AUC = 0.5). This trend is observed for both the "LeetCode +

Complexity	Model	Pre-Cutoff 1			Pre-Cutoff 2			Post-Cutoff			T-test 1 (Between pre1, pre2)		T-test 2 (Between pre2, post)	
		Avg	Std	Median	Avg	Std	Median	Avg	Std	Median	t-value	p-value	t-value	p-value
Easy	Llama-3.1-8B	0.59	0.07	0.60	0.58	0.08	0.58	0.35	0.00	0.35	0.704	0.4821	29.52	7.12E-51
	Qwen2.5-7B	0.79	0.06	0.80	0.79	0.06	0.79	0.38	0.00	0.38	0.709	0.4793	65.07	4.66E-73
	Qwen2.5-Coder	0.78	0.06	0.80	0.78	0.06	0.80	0.43	0.00	0.43	0.178	0.8587	58.206	2.20E-78
	DS-coder.6.7b	0.63	0.06	0.63	0.62	0.08	0.63	0.20	0.00	0.20	0.554	0.5802	54.219	2.00E-75
Medium	Llama-3.1-8B	0.25	0.07	0.25	0.24	0.06	0.25	0.05	0.03	0.05	0.359	0.7199	29.925	1.25E-61
	Qwen2.5-7B	0.55	0.08	0.55	0.54	0.08	0.55	0.08	0.03	0.08	1.053	0.2935	55.643	8.61E-96
	Qwen2.5-Coder	0.58	0.08	0.58	0.56	0.07	0.55	0.08	0.03	0.08	1.138	0.2566	60.466	3.99E-99
	DS-coder.6.7b	0.36	0.08	0.35	0.37	0.08	0.38	0.02	0.02	0.03	-0.711	0.4782	45.118	4.00E-73
Hard	Llama-3.1-8B	0.10	0.04	0.10	0.09	0.05	0.10	0.05	0.02	0.05	0.899	0.3696	8.738	7.25E-15
	Qwen2.5-7B	0.27	0.07	0.28	0.24	0.06	0.25	0.05	0.02	0.05	3.235	0.001426	28.802	4.79E-56
	Qwen2.5-Coder	0.29	0.07	0.29	0.28	0.07	0.28	0.06	0.02	0.08	1.457	0.1468	28.917	9.03E-56
	DS-coder.6.7b	0.17	0.06	0.15	0.16	0.06	0.15	0.02	0.01	0.03	1.214	0.2261	23.122	2.92E-43

Table 2: Pre-Post Cutoff Statistical Analysis

Model	LeetCode + Contamination Dataset			KodCode-V1 + Contamination Dataset		
	loss	min_k	zlib	loss	min_k	zlib
Llama-3.2-1.3B	0.55	0.55	0.54	0.51	0.53	0.48
Qwen2.5-0.5B	0.54	0.52	0.54	0.53	0.49	0.54
Qwen2.5-Coder-0.5B	0.54	0.54	0.54	0.49	0.55	0.54
DS-coder-1.3b-base	0.57	0.57	0.54	0.55	0.57	0.54

Table 3: AUC score of each post-hoc detection methods

Contamination Dataset" as well as the alternative contaminated dataset included in the table. These findings indicate that current post-hoc detection methods, regardless of the specific algorithm or underlying model, are largely ineffective at reliably distinguishing between contaminated and clean samples.

The weak performance of current membership inference attacks likely stems from several underlying factors intrinsic to large code models and the nature of code data. Unlike natural language, code exhibits strong regularity and redundancy: many syntactically distinct snippets can implement the same functionality. This structural redundancy causes similar loss distributions across both seen and unseen examples, making loss-based signals (central to most post-hoc attacks) less discriminative. Modern Code LLMs are also trained on massive, heterogeneous corpora with aggressive regularization, de-duplication, and instruction-tuning steps that blur the boundary between "memorization" and "generalization." As a result, the models often reproduce patterns learned from the broader distribution rather than exact memorized sequences, diminishing the sharp confidence gap MIAs rely on.

Answers to **RQ2**: Our results show that membership attack–based post-hoc methods are largely ineffective at detecting contamination in Code LLMs.

Complexity	Model	Leetcode		DyCodeEval + Pre	Delta	
		Pre	Post		Pre - Post	Pre - Dynamic Pre
Easy	Llama-3.1-8B	0.59	0.35	0.49	0.24	0.10
	Qwen2.5-7B	0.80	0.38	0.68	0.43	0.12
	Qwen2.5-Coder	0.79	0.43	0.69	0.37	0.10
	DS-coder.6.7b	0.63	0.20	0.64	0.43	0.00
Medium	Llama-3.1-8B	0.25	0.05	0.20	0.20	0.05
	Qwen2.5-7B	0.56	0.08	0.45	0.48	0.11
	Qwen2.5-Coder	0.58	0.08	0.46	0.50	0.12
	DS-coder.6.7b	0.40	0.03	0.38	0.37	0.01
Hard	Llama-3.1-8B	0.10	0.05	0.07	0.05	0.03
	Qwen2.5-7B	0.27	0.05	0.19	0.23	0.08
	Qwen2.5-Coder	0.29	0.06	0.21	0.23	0.08
	DS-coder.6.7b	0.17	0.02	0.17	0.16	0.01

Table 4: The dynamic benchmark effectiveness on Code generation tasks

4.3 RQ3 Results

The results in Table 4 and 5 provide a comprehensive view of the effectiveness of dynamic benchmarks for both code generation and code infilling tasks across various models and problem complexities.

Code Generation. Across all models and difficulty tiers, accuracy drops markedly from *LeetCode Pre* to *Post*, evidencing contamination in static benchmarks. Under dynamic evaluation (*DyCodeEval+Pre*), scores sit between *Pre* and *Post*: lower than *Pre* but consistently higher than *Post*, indicating reduced contamination while preserving task semantics. Crucially, the Pre→Post drop is much larger than the Pre→Dynamic-Pre drop (e.g., 0.50 vs 0.12 for Qwen2.5-Coder on medium), indicating that while dynamic benchmarks reduce contamination, some performance gap remains due to the models' reduced exposure to the test data.

Code Infilling. The same pattern holds: *Post* < *Pre* across complexities, with larger gaps on harder problems. Dynamic variants (Pre+IO, Pre+FW, Pre+IE) reduce scores relative to *Pre* yet remain above *Post* (Pre+IE typically closest). In most cases, Pre→Post declines exceed Pre→Dynamic-Pre, indicating that dynamic benchmarks, while

Complexity	Model	Leetcode (Infilling)					Delta			
		Pre	Post	Pre+IO	Pre+FW	Pre+IE	Pre - Post	Pre - (Pre + IO)	Pre - (Pre + FW)	Pre - (Pre +IE)
Easy	Llama-3.1-8B	0.755	0.725	0.678	0.616	0.775	0.030	0.076	0.139	-0.020
	Qwen2.5-7B	0.880	0.700	0.740	0.568	0.755	0.180	0.140	0.312	0.125
	Qwen2.5-Coder	0.869	0.775	0.841	0.753	0.823	0.094	0.028	0.116	0.046
	DS-coder.6.7b	0.710	0.413	0.646	0.636	0.630	0.297	0.064	0.074	0.080
Medium	Llama-3.1-8B	0.693	0.565	0.636	0.471	0.715	0.128	0.057	0.222	-0.022
	Qwen2.5-7B	0.767	0.612	0.600	0.429	0.602	0.155	0.167	0.338	0.165
	Qwen2.5-Coder	0.797	0.675	0.780	0.599	0.836	0.122	0.017	0.198	-0.039
	DS-coder.6.7b	0.562	0.313	0.513	0.510	0.512	0.249	0.050	0.052	0.050
Hard	Llama-3.1-8B	0.559	0.565	0.548	0.334	0.605	-0.005	0.011	0.225	-0.045
	Qwen2.5-7B	0.613	0.573	0.492	0.283	0.513	0.040	0.121	0.330	0.100
	Qwen2.5-Coder	0.655	0.566	0.717	0.407	0.695	0.089	-0.062	0.248	-0.040
	DS-coder.6.7b	0.355	0.216	0.300	0.283	0.333	0.139	0.055	0.072	0.023

Table 5: The dynamic benchmark effectiveness on code infilling tasks

stricter, still yield smaller drops than truly novel (post-cutoff) data.

> Answers to **RQ3**: Our experiments show that dynamic benchmarking can partially mitigate contamination, but its effectiveness is inconsistent and requires further study.

5 Related Work

Data Contamination in LLMs. As large language models (LLMs) continue to improve in accuracy and performance, concerns have grown regarding data contamination: the inadvertent inclusion of benchmark or test data in the training corpus. This can lead to artificially inflated performance metrics and undermine the validity of evaluations, creating a false sense of model progress and potentially misleading both researchers and practitioners. As a result, much recent work (Chen et al., 2025a; Golchin and Surdeanu, 2024) has focused on understanding, detecting, and mitigating data contamination in LLM training and evaluation pipelines.

Several approaches have been proposed to address this issue. For example, LiveCodeBench (Jain et al., 2024) leverages model cutoff dates, comparing accuracy across different release periods to identify performance drops after the cutoff. However, this approach does not control for problem difficulty—a significant confounding factor when comparing accuracy across time.

Other efforts rely on membership-inference attacks (MIA) to detect contamination. One such method, Zlib Entropy (Carlini et al., 2021), measures the compressibility of a sequence and compares this against model perplexity to infer whether a sample was present in the training data. While promising, these attacks have not been systematically tested on software engineering tasks. To the best of our knowledge, our work is the first to extend contamination detection methods to this domain.

Contamination Prevention Efforts. Beyond detection, substantial effort has also been devoted to preventing contamination. One line of work focuses on protecting benchmarks through encryption and privacy-preserving mechanisms (Rajore et al., 2024; Jacovi et al., 2023), ensuring that test data cannot be easily incorporated into training corpora or reverse-engineered. This remains a difficult challenge, however, as publicly available benchmarks are both a cornerstone of the field and a driver of rapid model development. Another approach is the use of dynamic evaluation frameworks, such as DyVal, which adaptively generate new or mutated test data at evaluation time to minimize overlap with training corpora and provide a more robust measure of generalization (Chen et al., 2025a, 2024; Zhu et al., 2024; Yu et al., 2024b; Wang et al., 2022). By keeping benchmarks fresh and continually evolving, these frameworks make it more difficult for LLMs to rely on memorization or exploit benchmark leakage.

6 Conclusion

We present the first in-depth empirical study of data contamination in Code LLMs and evaluate existing detection and mitigation methods. Our results show that current models memorize benchmark data, membership inference attacks fail to detect contamination in Code LLMs, and dynamic benchmarking faces practical limitations. These findings reveal a gap between leaderboard performance and true generalization, highlighting the need for robust evaluation tools, contamination-free benchmarks, and transparent data curation. Our study lays the groundwork for more reliable and trustworthy assessment of future Code LLMs.

Limitations

Our proposed work has several limitations:

Model scale: Due to computational constraints, we were only able to fine-tune smaller LLMs and evaluate MIA attacks on them. Since larger models often exhibit different memorization behaviors, we plan to extend our analysis to larger-scale models in future work.

Task coverage: To highlight signals of data contamination in software engineering, our experiments focused primarily on code generation and infilling. We aim to broaden this scope to additional tasks such as code translation and bug fixing to provide a more comprehensive evaluation.

Assumptions about contamination: In performing MIA attacks on our base models, we assumed that the fine-tuning and test problems were free of prior contamination. However, slight leakage may exist, and addressing this rigorously would require retraining models from scratch—an expensive process that was beyond the scope of this study.

References

Rachith Aiyappa, Jisun An, Haewoon Kwak, and Yong-yeol Ahn. 2023. Can we trust the evaluation on ChatGPT? In *Proceedings of the 3rd Workshop on Trustworthy Natural Language Processing (TrustNLP 2023)*, pages 47–54, Toronto, Canada. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Jialun Cao, Wuqi Zhang, and Shing-Chi Cheung. 2024. Concerned with data contamination? assessing countermeasures in code language model. *Preprint*, arXiv:2403.16898.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting training data from large language models. *Preprint*, arXiv:2012.07805.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Simin Chen, Yiming Chen, Zexin Li, Yifan Jiang, Zhongwei Wan, Yixin He, Dezhi Ran, Tianle Gu, Haizhou Li, Tao Xie, and 1 others. 2025a. Recent advances in large langauge model benchmarks against data contamination: From static to dynamic evaluation. *arXiv preprint arXiv:2502.17521*.

Simin Chen, Xiaoning Feng, Xiaohong Han, Cong Liu, and Wei Yang. 2024. Ppm: Automated generation of diverse programming problems for benchmarking code generation models. *Proceedings of the ACM on Software Engineering*, 1(FSE):1194–1215.

Simin Chen, Pranav Pusarla, and Baishakhi Ray. 2025b. Dynamic benchmarking of reasoning capabilities in code large language models under data contamination. *Preprint*, arXiv:2503.04149.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, and 21 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

Michael Duan, Anshuman Suri, Niloofar Mireshghallah, Sewon Min, Weijia Shi, Luke Zettlemoyer, Yulia Tsvetkov, Yejin Choi, David Evans, and Hannaneh Hajishirzi. 2024. Do membership inference attacks work on large language models? *Preprint*, arXiv:2402.07841.

Lizhou Fan, Wenyue Hua, Xiang Li, Kaijie Zhu, Mingyu Jin, Lingyao Li, Haoyang Ling, Jinkui Chi, Jindong Wang, Xin Ma, and Yongfeng Zhang. 2024. Nphardeval4v: A dynamic reasoning benchmark of multimodal large language models. *Preprint*, arXiv:2403.01777.

Shahriar Golchin and Mihai Surdeanu. 2024. Time travel in llms: Tracing data contamination in large language models. *Preprint*, arXiv:2308.08493.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Yu He, Boheng Li, Liu Liu, Zhongjie Ba, Wei Dong, Yiming Li, Zhan Qin, Kui Ren, and Chun Chen. 2025. Towards label-only membership inference attack against pre-trained large language models. *Preprint*, arXiv:2502.18943.

Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. 2023. Stop uploading test data in plain

text: Practical strategies for mitigating data contamination by evaluation benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5075–5084, Singapore. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Yucheng Li, Frank Guerin, and Chenghua Lin. 2024. Latesteval: Addressing data contamination in language model evaluation through dynamic and time-sensitive test construction. *Preprint*, arXiv:2312.12343.

Ziyu Li and Donghwan Shin. 2024. Mutation-based consistency testing for evaluating the code understanding capability of llms. *Preprint*, arXiv:2401.05940.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, and 31 others. 2023. Holistic evaluation of language models. *Preprint*, arXiv:2211.09110.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Preprint*, arXiv:2305.01210.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2025. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Inbal Magar and Roy Schwartz. 2022. Data contamination: From memorization to exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 157–165, Dublin, Ireland. Association for Computational Linguistics.

Tanmay Rajore, Nishanth Chandran, Sunayana Sitaram, Divya Gupta, Rahul Sharma, Kashish Mittal, and Manohar Swaminathan. 2024. Truce: Private benchmarking to prevent contamination and improve comparative evaluation of llms. *Preprint*, arXiv:2403.00393.

Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. 2024. To the cutoff... and beyond? a longitudinal perspective on llm data contamination. In *Proceedings of ICLR 2024*.

Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. 2024. Detecting pretraining data from large language models. *Preprint*, arXiv:2310.16789.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2022. Recode: Robustness evaluation of code generation models. *Preprint*, arXiv:2212.10264.

Siyuan Wang, Zhuohan Long, Zhihao Fan, Zhongyu Wei, and Xuanjing Huang. 2024. Benchmark self-evolving: A multi-agent framework for dynamic llm evaluation. *Preprint*, arXiv:2402.11443.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024a. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. *Preprint*, arXiv:2312.14187.

Zhuohao Yu, Chang Gao, Wenjin Yao, Yidong Wang, Wei Ye, Jindong Wang, Xing Xie, Yue Zhang, and Shikun Zhang. 2024b. Kieval: A knowledge-grounded interactive evaluation framework for large language models. *Preprint*, arXiv:2402.15043.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2024. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *Preprint*, arXiv:2303.17568.

Kaijie Zhu, Jindong Wang, Qinlin Zhao, Ruochen Xu, and Xing Xie. 2024. Dynamic evaluation of large language models by meta probing agents. *Preprint*, arXiv:2402.14865.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *Preprint*, arXiv:2406.15877.

A Dataset Description

Here is a brief description of each dataset used in the experiments described in the paper:

1. **`HumanEval:`** An open-source benchmark consisting of 164 Python programming problems released by OpenAI. Each problem includes a prompt (comprising of import statements, a function signature, and a natural language description) along with a canonical solution and corresponding test cases.

2. **`MBPP-Sanitized:`** A crowd-sourced dataset of 427 basic Python programming problems. Each problem has been reformatted to include the task description within the function docstring, aligning with the HumanEval format.

3. **`HumanEval Infilled:`** A single-line infilled variant of HumanEval. Each HumanEval problem is decomposed into n subproblems, where n corresponds to the number of lines in the original solution. In each subproblem, one line from the solution is removed. Each subproblem includes a prefix (containing the function signature and problem description), a suffix, the target line to be predicted (i.e., the missing line), and accompanying test cases.

4. **`MBPP-Sanitized Infilled:`** A single-line infilled version of MBPP-Sanitized.

5. **`Leetcode Dataset:`** A total of 2,869 Python programming problems collected from the LeetCode platform. Each problem was modified to include all relevant import statements used during code execution. The prompt combines the original problem description with the corresponding starter code provided by LeetCode.

6. **`Leetcode Infilled Dataset:`** A single-line infilled version of Leetcode Dataset.

B RQ1 Additional Results

In RQ1, we conducted studies comparing Leetcode pre and post model cutoff problems. We accounted for difficulty of problem using 3 factors: Leetcode manually labeled complexity, input token length, and solution line length. Here are the results for the input token length and solution line length:

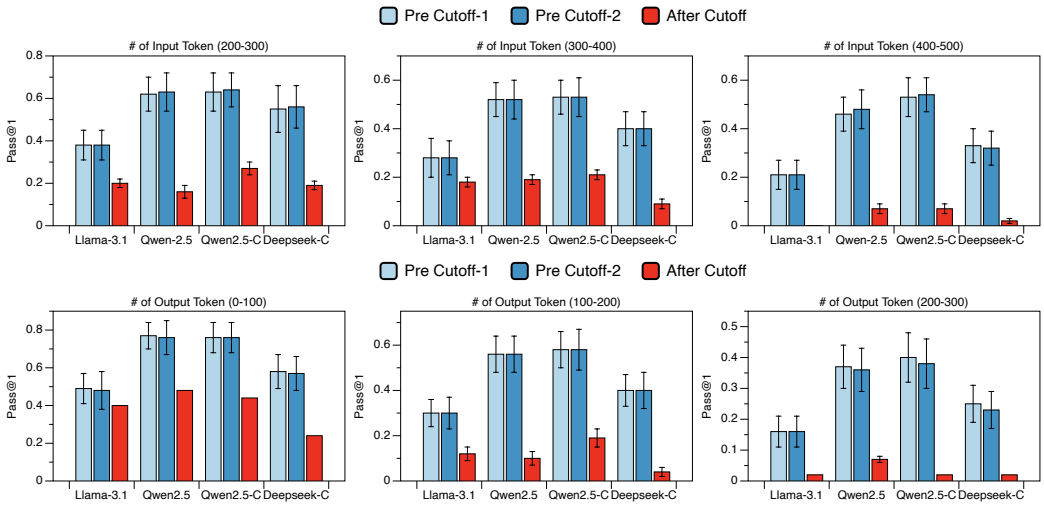

Figure 6: Code Generation Benchmarking Results for Token Complexity

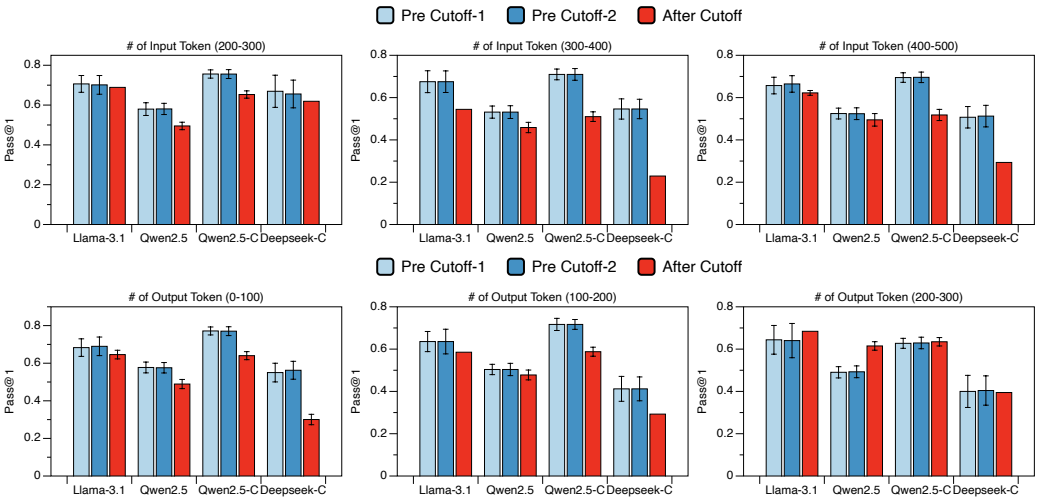

Figure 7: Code Infilling Benchmarking Results For Token Complexity