Is Your Benchmark Still Useful? Dynamic Benchmarking for Code Language Models

Anonymous Author(s)
Affiliation
Address
email

Abstract

In this paper, we tackle a critical challenge in model evaluation: how to keep code benchmarks useful when models might have already seen them during training. We introduce a novel solution, dynamic benchmarking framework, to address this challenge. Given a code understanding or reasoning benchmark, our framework dynamically transforms each input, *i.e.*, programs, with various semantic-preserving mutations to build a syntactically new while semantically identical benchmark. We evaluated 10 popular language models on our dynamic benchmarks. Our evaluation reveals several interesting or surprising findings: (1) all models perform significantly worse than before, (2) the ranking between some models shifts dramatically, and (3) dynamic benchmarks can resist against the data contamination problem.

1 Introduction

During the past period, the emergence of advanced Large Language Models (LLMs) such as the GPT [27], Llama [6], and DeepSeek [5] series have revolutionized the field of software development through their exceptional performance in various code-related tasks, including but not limited to code generation [17], debugging [21], and optimization [16]. Rapid integration of LLM-driven development tools and plugins into mainstream programming environments, exemplified by GitHub Copilot [7] and Cursor [4], not only validates the practical utility of these models, but also significantly enhances the productivity of developers in the software industry. This widespread adoption ultimately underscores the transformative potential of LLMs in code understanding and reasoning, where LLMs are expected to comprehend the underlying semantics in the input code [24].

During the rapid advancement of LLMs, the community has developed various benchmarks to evaluate the code reasoning capabilities of the model. For instance, CRUXEval [9] specifically targets code execution proficiency, while Code Lingua [28] and TransCoder [19, 33] focus on evaluating code translation skills. The successful completion of these tasks requires LLMs to demonstrate accurate comprehension of code semantics; hence, these benchmarks serve as effective indicators of the LLMs' code reasoning capabilities to a certain extent.

There are a few common practices to collect code for a benchmark. Some benchmarks automatically crawl code from public databases such as GitHub [15, 26], while others take advantage of advanced LLMs, such as ChatGPT, to generate new code [9]. In some cases, human experts are involved in the manual writing of the code. Regardless of how the code is collected, all these open-source benchmarks will eventually be available to the public for fair model evaluation. Modern LLMs are trained on large amounts of data, both public and private. For instance, Qwen2.5-Coder's report states that they collect publicly available repositories on GitHub for training and remove key datasets using a 10-gram overlap method [15]. However, removing key datasets does not guarantee that all data used for evaluation is eliminated, and there is still a risk of data contamination. This raises a crucial concern:

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

```
def f(lists):
    lists[1].clear()
    lists[2] += lists[1]
    return lists[0]
assert f([[395, 666, 7, 4], [],\
[4223, 111]]) == [395, 666, 7, 4]
                 [395, 666, 7, 4]
```

(a) Original Version

```
def f(var1):
    var1[1].clear()
    var1[2] += var1[1]
    return var1[0]
assert f([[395, 666, 7, 4], [],\
[4223, 111]]) == 395
                 [395, 666, 7, 4]
```

(b) Mutated Version

Figure 1: Answers produced by GPT-4o mini and DeepSeek V3. The modified variables are highlighted in blue for identification.

Are these benchmarks still useful for evaluating models if they are unavoidably included in the training data?

Take Figure 1 as an example. On the left is a test case from the CRUXEval benchmark [9]. The test case consists of a function `f` together with the input `([[395, 666, 7, 4], [], [4223, 111]])` — a list of lists — encoded in an `assert`. Models are tasked to predict the function's output. The function `f` executes a sequence of operations: first erasing the value in `lists[1]`, then adding this value to `lists[2]`, and finally outputting the value of `lists[0]`. GPT-4o mini successfully emits the correct output in this scenario. However, as shown in Figure 1(b), after we change the variable name into `var1`, GPT-4o mini produces an incorrect result with even the wrong type (an integer instead of a list of integers). This behavioral shift suggests that in Figure 1(a), the descriptive variable name `lists` may have assisted the model's comprehension, potentially due to the contamination of variable name `lists` in the training data. In Figure 1(b), however, the absence of such hints, combined with the common pattern of `var1[0]` typically representing integer indices, likely led the model to erroneously infer an integer return type. DeepSeek V3, on the other hand, successfully handles both cases, indicating its reasoning beyond variable names. Nevertheless, the original benchmark fails to differentiate these two models' reasoning capabilities with this test case.

When the code in Figure 1(a) appears in the training set of a model, such code cannot be effectively used to measure the code understanding and reasoning capability of the model. Unfortunately, publicly accessible benchmarks are persistently plagued by the issue of training data contamination, which fundamentally compromises the integrity and reliability of evaluation results [1]. This phenomenon is particularly problematic given the remarkable memorization capacity of LLMs [11], where models may have already been exposed to substantial portions of the evaluation data[1], which are open source online, during their training phase. Such exposure enables models to potentially reproduce or closely approximate memorized patterns rather than demonstrating genuine reasoning or generalization capabilities. Consequently, this leads to inflating performance metrics that fail to accurately reflect the models' true ability to handle novel, unseen data, thereby undermining the validity of benchmark comparisons and the meaningful assessment of model capabilities.

In this paper, we present dynamic benchmarking[2], a new benchmarking framework to address the aforementioned challenge. Our framework takes an established code benchmark and mutates each test case with a set of semantic-preserving mutations. All the mutated code snippets are syntactically different from the original benchmark while having identical code semantics. For instance, Figure 1(b) is a mutant of Figure 1(a) — despite being syntactically different, both test cases have identical execution behaviors. We instantiate our framework using a set of mutations that work on both the *code syntax* and *code structure* levels. At the code syntax level, we focus on variable naming, *i.e.*, normalizing or randomizing variable names, to remove irrelevant assistance information from them.[3] At the code structure level, we focus on three core code structures, *i.e.*, assignment, conditional branching, and loops. By requiring models to process these mutated versions, we ensure that the evaluation accurately measures the model's capacity for code understanding and reasoning, rather than its ability to recall specific code patterns from its training data.

[1]Or code corpus of similar syntactic patterns.

[2]Released at: `https://www.kaggle.com/datasets/aspartic/dynamic-benchmark`

[3]We *do not* use adversarially mutated variable names to deliberately mislead models.

We evaluate our mutations on four benchmarks covering two coding tasks. The results show that in our new benchmarks, all models' performance declines significantly, up to 40% lower than the original benchmarks. In order to evaluate whether or not our dynamic benchmarking approach can resist the data contamination problem, we fine-tuned a model with the original benchmark. The results show that our dynamically constructed benchmarks are moderately affected and are still useful in evaluating the fine-tuned model.

2 Methodology

2.1 Dynamic Benchmarking Framework

LLM-based code reasoning generates outcomes based on high-level instructions and source code. Typically, the instructions describe the specific reasoning task in natural language, while the source code represents the program to be analyzed. Generally, we expect the LLM to derive results through reasoning based on the provided instructions and source code. Formally, for a given benchmark with n test cases $\{< ins_i, p_i, o_i >\}, i \in [1, n]$ and a model LLM under evaluation. The benchmark evaluates how correct $LLM(ins_i, p_i)$ is compared to the reference result o_i. Ideally, all $\{< ins_i, p_i, o_i >\}$ should be unknown to the model. However, due to the data contamination problem, $\{< ins_i, p_i, o_i >\}$ are often leaked.

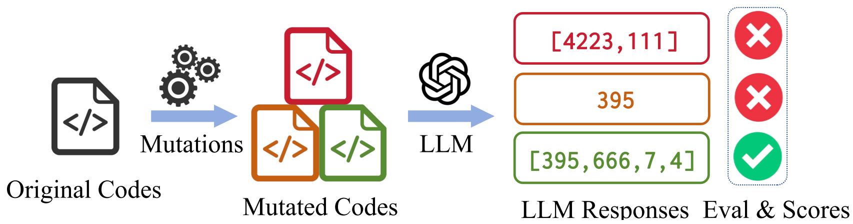

Figure 2: The dynamic benchmarking framework.

Figure 2 shows the high-level workflow of our dynamic benchmarking framework. Given one $< ins_i, p_i, o_i >$, our framework applies a set of semantics-preserving mutations M_k to transform the test case into multiple ones as follows:

$$< ins_i, p'_i = M_{k_1}(\cdots M_{k_m}(p_i)), o_i >,$$

where M_{k_j} is one of the available mutations. The new program p'_i is sequentially mutated from the original program p. Since all M_{k_j} are semantics-preserving, the final reference result is still o_i. All the new test cases are dynamically generated from the existing benchmark and are then used for evaluating models. To ensure the usability of newly generated test cases, each mutation method is governed by the following two fundamental requirements:

Semantic Equivalence. The mutated program p' must maintain execution behavior identical to the original program p. Formally, for all possible inputs x of program p, the following condition should be satisfied:

$$\forall x, [[p]]_x = [[M_{k_j}(p)]]_x,$$

where $[[p]]_x$ denotes the execution result of program p on the input x. This requirement guarantees *the mutation preserves the program's semantics*.

Syntactic Divergence. The mutated program p' should be syntactically different from the original program p in a non-trivial manner. This property ensures that even if p is included in the training set of a model, p' can still be used to evaluate the model.

In the following sections, we will propose a series of mutation methods at two levels: *code syntax* and *code structure*.

2.2 Code Syntax Mutations

In the process of models' reasoning about code, conventional static benchmarks typically employ meaningful variable names, which inadvertently provide models with additional cues that may influence their reasoning process. This phenomenon allows models to potentially utilize syntactic sugar information, particularly through variable naming patterns, as auxiliary features to facilitate

```python
def f(lst: list):
    length = len(lst)
    for i in range(length):
        if i == 3:
            lst[i] = 5
    return lst
```

(a) Original Code

```python
def f(var1: list):
    var2 = len(var1)
    for var3 in range(var2):
        if var3 == 3:
            var1[var3] = 5
    return var1
```

(b) Variable Normalization I

```python
def f(QxL: list):
    Rjm = len(QxL)
    for kpN in range(Rjm):
        if kpN == 3:
            QxL[kpN] = 5
    return QxL
```

(c) Variable Normalization II

```python
def f(lst: list):
    length = len(lst)
    for i in range(length):
        if i == 3:
            lst[i] = 7 - 2
    return lst
```

(d) Constant Unfolding

```python
def f(lst: list):
    length = len(lst)
    i = 0
    while i < length:
        if i == 3:
            lst[i] = 5
        i += 1
    return lst
```

(e) For to While

```python
def f(lst: list):
    length = len(lst)
    for i in range(length):
        if i == 3 and \
         ((8 > 6)or(8 < 6)):
            lst[i] = 5
    return lst
```

(f) Condition Augmentation

Figure 3: Illustrative examples of our code syntax and code structure mutations. (a) is the code from the original benchmark, while (b)-(f) are new codes by applying one of our mutations.

code comprehension and reasoning. As demonstrated in Figure 1, models can leverage semantically rich variable names to infer program functionality without fully grasping the underlying algorithmic logic. Moreover, once the static benchmark is leaked into the training set, variable names will become convenient objects that can be easily memorized.

Variable Normalization. To address these critical issues, we propose implementing *Variable Normalization* as a methodological solution to build a dynamic benchmark. Specifically, we implement two distinct normalization schemes: ***VarNormI***, a sequential naming convention (e.g., `var1`, `var2`, `var3`, ...) that maintains variable uniqueness while eliminating semantic cues, and ***VarNormII***, a fixed-length randomized string generation system that ensures complete obfuscation of any potential naming-related information. Figure 3(a) presents a piece of original code with a format similar to CRUXEval. Figure 3(b) and Figure 3(c) are the new mutants after applying VarNormI and VarNormII mutations. As the figure illustrates, variable normalization effectively eliminates the syntactic sugar information embedded in descriptive variable names such as `lists` and `length`. By obfuscating the semantic information contained in variable names, this technique effectively forces LLMs to focus on the underlying structural and logical relationships within the code rather than relying on superficial naming patterns. Furthermore, since variable normalization exclusively modifies variable names without altering the underlying code structure, it inherently guarantees both semantic equivalence and syntactic divergence between the original and mutated code.

2.3 Code Structure Mutation

Understanding code involves more than just syntactic information. It also involves understanding how data moves through code (data flow) and how it makes decisions (control flow). To address data contamination in such code structures, we present a structural transformation approach for dynamic benchmark construction. Our approach systematically changes the structure of code samples while preserving their semantic equivalence and functional integrity. Specifically, we focus on three fundamental and ubiquitous code structures that represent core programming constructs: *Assignment*, *Loop*, and *Branch*. We employ Python syntax as our demonstration framework to systematically illustrate the implementation of our mutation methodology.

Assignment. Our analysis specifically targets the fundamental case of constant variable assignment, as exemplified by basic statements like `a = 5`. To address this, a mutation technique termed ***Constant Unfolding*** has been developed, which systematically decomposes constants (e.g., integers) into equivalent arithmetic expressions while rigorously preserving semantic equivalence throughout the transformation process. This transformation is a reversion of the classical *constant folding* optimization in compiler design [31]. As demonstrated in Figure 3(d), *Constant Unfolding* transforms the assignment statement `lists[i] = 5` into `lists[i] = 7 - 2`. This mutation method only transforms an integer to an expression, so the semantic equivalence is maintained. Moreover, the proportion of text requiring modification remains minimal. By transforming straightforward integer assignments into equivalent arithmetic expressions, we create a subtle yet effective barrier against simple pattern recognition while preserving the essential reasoning challenges.

Loop. In the domain of iterative control structures, two primary loop paradigms exist: *for* loops and *while* loops. These loop constructs, while syntactically distinct, achieve comparable computational complexity and can be mutually transformed through systematic restructuring. Our mutation framework incorporates the ***For to While*** strategy, which transforms all *for* loops into functionally equivalent *while* loops and vice versa, as exemplified in Figure 3(e). This mutation strategy also meets our requirements for mutation methods. Firstly, the semantic equivalence between *for* loops and *while* loops is guaranteed due to their interchangeable nature in representing iterative processes. This transformation maintains identical program behavior while introducing syntactic variations. Secondly, the modification scope is significantly limited, as only the loop header requires alteration.

Branch. Within conditional branching structures, we implement a transformation technique called ***Condition Augmentation***, which introduces tautological expressions into existing conditions while preserving the original logical outcome. At a high level, given an *if* condition, it can always be equivalently transformed in the following way:

$$if(C) \Leftrightarrow if(C \text{ \& True}) \Leftrightarrow if(C \text{ || False})$$

Guided by this principle, our implementation instantiates the condition *True* or *False* with predetermined yet complex expressions. For example, as illustrated in Figure 3(f), we augment the condition `i == 3` by incorporating a tautological expression `(8 > 6) or (8 < 6)`, resulting in the modified condition `i == 3 and ((8 > 6) or (8 < 6))` while maintaining the original logical equivalence. In addition, there are some optional tautology expressions, such as `True or False`. In actual experiments, we will use these tautology expressions in combination. The mutation approach solely introduces tautological conditions in the branching part, leaving the code semantics unaffected. Thus, it meets our requirements.

Discussion: The "naturalness" of mutated code. We deliberately exclude considerations of code "naturalness" in our evaluation framework for several reasons. First, existing naturalness metrics, such as language model likelihoods [18, 32], are inherently biased towards their training data distributions and fail to capture the nuanced aspects of code quality, such as idiomatic usage, coding conventions, and style consistency. More fundamentally, there exists no precise or universally accepted metric for evaluating code naturalness, making such assessments inherently subjective and potentially misleading. This limitation is particularly relevant given that many benchmarks, such as CRUXEval's test functions which are generated using Code Llama, are deliberately constructed rather than derived from real-world code. In this context, the pursuit of naturalness becomes a secondary concern. For code language models, the true value of a benchmark lies in its ability to rigorously evaluate a model's fundamental capabilities: its understanding of code semantics and its capacity for logical reasoning, rather than the superficial syntax.

2.4 Multi-Mutation

All the aforementioned mutation methods operate independently of each other. As a result, these methods can be combined for use, enabling multi-mutation. This combined approach still satisfies our two key requirements for mutation methods. Firstly, since none of the individual mutations affect the program's execution outcome, multi-mutation likewise preserves the program's original behavior. Secondly, while individual mutations introduce minimal changes to the code text, the cumulative impact of multi-mutation remains limited and manageable. Therefore, this approach can be utilized effectively to build dynamic benchmarks and further test the code reasoning capabilities of LLMs.

3 Experimental Setup

Our evaluation aims to answer the following Research Questions (RQs):

1. (**Impact of single mutation**) How does models' performance change under each single mutation?
2. (**Impact of multi-mutation**) How does models' performance change under multi-mutation?
3. (**Mitigation of data contamination**) Can dynamic benchmarking mitigate data contamination?
4. (**Complexity of dynamic benchmarks**) How much additional complexity does our approach introduce to the original benchmarks?

We selected a variety of both closed and open-source LLMs. Specifically, we chose models including GPT-4o mini [27], DeepSeek V3 [5], Llama 3.1 series [8], Qwen2.5-Coder series [14], and StarCoder2 series [26]. Our models are sourced from the APIs provided by OpenRouter[4] and the open-source models available on Hugging Face[5]. In our experiments, we select two popular tasks: *code execution* and *code translation*. For code execution, we selected the CRUXEval benchmark [9], while for code translation, we selected the Code Lingua benchmark (which includes two sub-datasets, Avatar and CodeNet) [28] and the TransCoder benchmark [19, 33], on which we conduct experiment on Python to Java translation.

For all LLMs, we set the temperature to 0.2 to evaluate their Pass@1 metric [2]. We generate 5 results for each sample in the benchmarks. On the aforementioned four datasets, we applied five distinct mutation methods to each program within the datasets. These methods, as outlined in Section 2, include two types of Variable Normalization (**VarNormI** and **VarNormII**), Constant Unfolding (**ConstUnfold**), For to While (**For2While**), and Condition Augmentation (**CondAug**).

As for multi-mutation, since we have five different mutation methods, there are dozens of combinations in theory. For a practical evaluation overhead, we randomly select three mutation combinations that cover all mutation methods from different categories: **FUV** (For2While (F) + ConstUnfold (U) + VarNormII (V)), **AUV** (CondAug (A) + ConstUnfold (U) + VarNormI (V)) and **AFU** (CondAug (A) + For2While (F) + ConstUnfold (U)).

4 Results and Analysis

4.1 RQ1: Impact of Single Mutation

Due to space constraints, we present only the model Pass@1 performance results on both original static and our dynamic CRUXEval and CodeNet benchmarks in the main text. Table 1 shows the results.

General Tendency. The table reveals a consistent performance degradation tendency across most mutations, indicating that models demonstrate significantly reduced comprehension capabilities when processing mutated code compared to their performance on original code. For instance, after applying the Constant Unfolding method, the performance of many models dropped by more than 10% compared to their performance on the original benchmark. This significant decline clearly demonstrates that models exhibit reduced code reasoning capabilities on dynamic benchmarks, thereby validating the effectiveness of our approach. Moreover, we observe that in a small number of cases, models exhibit slightly improved reasoning capabilities on the mutated benchmark compared to the original benchmark, particularly in the StarCoder2 series. This phenomenon may be attributed to the fact that the StarCoder2 models are not instruction-tuned, resulting in weaker adherence to instructions. As a consequence, their performance on the benchmark may not fully reflect their potential and is more susceptible to randomness, leading to such reversals.

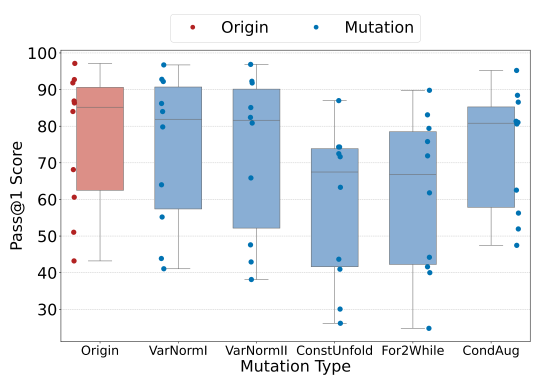

Figure 4: Comparison of model performance distributions under different mutations of CodeNet.

Benchmark Effectiveness Enhancement. The significant performance drop of models on the dynamic benchmark also enhances the usability of certain benchmarks. For example, as shown in Table 1, on the original CodeNet benchmark, the GPT-4o mini model achieves a performance of 90.20%, indicating that the original CodeNet benchmark is relatively simple and no longer suitable for evaluating highly advanced models. However, after applying the Constant Unfolding mutation, the Pass@1 performance of GPT-4o mini drops to 72.49%. This demonstrates that our method makes this benchmark useful again by reintroducing meaningful differentiation among state-of-the-art models.

[4]https://openrouter.ai/

[5]https://huggingface.co/

Table 1: Model Pass@1 performance on original and single mutated CRUXEval and CodeNet benchmarks. Performance drops of more than 10% are highlighted.

Type	Model	Origin	VarNormI	VarNormII	ConstUnfold	For2While	CondAug
CRUXEval	DeepSeek V3	65.80	63.91 (-1.89)	63.01 (-2.79)	46.70 (**-19.10**)	58.01 (-7.79)	63.07 (-2.73)
	GPT-4o mini	53.60	51.80 (-1.80)	52.08 (-1.52)	34.51 (**-19.09**)	47.61 (-5.99)	54.71 (-3.89)
	Llama 3.1 70B	53.20	51.21 (-1.99)	51.94 (-1.26)	34.95 (**-18.25**)	45.95 (-7.25)	53.10 (-0.10)
	Llama 3.1 8B	31.20	30.10 (-1.10)	28.46 (-2.74)	21.80 (-9.40)	20.46 (**-10.74**)	23.96 (-7.24)
	Qwen2.5 32B	65.50	63.59 (-1.91)	63.02 (-2.48)	40.26 (**-25.24**)	57.22 (-8.28)	60.43 (-5.07)
	Qwen2.5 14B	58.30	57.27 (-1.03)	57.10 (-1.20)	39.21 (**-19.09**)	53.01 (-5.29)	58.98 (+0.68)
	Qwen2.5 7B	45.70	43.77 (-1.93)	43.97 (-1.73)	26.55 (**-19.15**)	42.03 (-3.67)	42.14 (-3.56)
	StarCoder2 15B	33.10	12.43 (**-20.67**)	17.07 (**-16.03**)	21.45 (**-11.65**)	31.37 (-1.73)	29.20 (-3.90)
	StarCoder2 7B	33.80	24.64 (-9.16)	25.27 (-8.53)	20.84 (**-12.96**)	33.27 (-0.53)	32.94 (-0.86)
	StarCoder2 3B	30.00	25.32 (-4.68)	24.94 (-5.06)	20.79 (-9.21)	27.97 (-2.03)	31.76 (+1.76)
CodeNet	Deepseek V3	94.70	96.73 (+2.03)	96.87 (+2.17)	86.98 (-7.72)	89.80 (-4.90)	95.22 (+0.52)
	GPT-4o mini	90.20	92.20 (+2.00)	91.80 (+1.60)	72.49 (**-17.71**)	83.10 (-7.10)	88.43 (-1.77)
	Llama 3.1 70B	88.00	79.80 (-8.20)	80.86 (-7.14)	74.32 (**-13.68**)	75.80 (**-12.20**)	81.34 (-6.66)
	Llama 3.1 8B	66.70	64.00 (-2.70)	65.87 (-0.83)	43.67 (**-23.03**)	44.20 (**-22.50**)	56.27 (**-10.43**)
	Qwen2.5 32B	92.70	92.80 (+0.10)	92.27 (-0.43)	74.32 (**-18.38**)	79.40 (**-13.30**)	86.57 (-6.13)
	Qwen2.5 14B	86.30	86.20 (-0.10)	85.09 (-1.21)	71.66 (**-14.64**)	71.90 (**-14.40**)	80.60 (-5.70)
	Qwen2.5 7B	84.10	84.00 (-0.10)	82.40 (-1.70)	63.31 (**-20.79**)	61.80 (**-22.30**)	81.04 (-3.06)
	StarCoder2 15B	58.60	55.20 (-3.40)	47.60 (**-11.00**)	40.95 (**-17.65**)	41.60 (**-17.00**)	62.54 (+3.94)
	StarCoder2 7B	43.00	41.07 (-1.93)	38.13 (-4.87)	26.18 (**-16.82**)	40.00 (-3.00)	47.46 (+4.46)
	StarCoder2 3B	50.70	43.87 (-6.83)	42.93 (-7.77)	30.06 (**-20.64**)	24.80 (**-25.90**)	51.94 (+1.24)

Performance Differentiation. Moreover, the dynamic benchmark enhances the differentiation of performance among various models by applying mutations to the benchmark. Figure 4 presents box plots illustrating the performance distribution of different models on the CodeNet benchmark before and after mutation. As shown in the figure, the performance distribution of models after mutation is significantly more spread out compared to the original static benchmark, which facilitates a better distinction between the capabilities of different models. This is also evident in specific examples. On the original CodeNet benchmark, the DeepSeek V3 model achieves a performance of 94.70%, while the GPT-4o mini model reaches 90.20%, indicating a relatively small gap between the two. However, after applying the Constant Unfolding mutation, the Pass@1 performance of DeepSeek V3 drops to 86.98%, whereas GPT-4o mini's performance declines significantly to 72.49%, resulting in a much larger gap. This demonstrates that the dynamic benchmark enables a more effective evaluation of the performance differences among models.

4.2 RQ2: Impact of Multi-Mutation

Here, we present a comparison between static and dynamic CRUXEval and CodeNet benchmarks based on multi-mutation, as shown in Table 2. By combining different mutation methods, the results show some new variations compared to using a single mutation method. We will explain these changes in detail below.

Drastic Performance Drop. From the table, it can be observed that the performance drop caused by multi-mutation is more significant than that of a single mutation. On CRUXEval, the performance drop can reach *as high as 20% or even close to 30%*. On CodeNet, many models incur *30% to 50%* performance drop. These phenomena indicate that our method achieves even better results when applied in combination.

Ranking Changes. In Table 1, although the absolute performance of the models has declined, the relative ranking changes are not significant. However, after applying multi-mutation, the relative rankings of the models have undergone observable changes. For example, in the CodeNet benchmark, the rankings of the three models from the Qwen2.5-Coder series experienced significant changes. The original ranking *is 32B >14B >7B*, but after applying the FUV method, the ranking *becomes 14B >32B >7B*, and with the AUV method, the ranking *shifts to 7B >14B >32B*. These ranking changes also suggest that, when using the original static benchmark, the inference performance of these models might not have been accurately evaluated due to issues like data contamination. Additionally, the CodeNet Benchmark was released in January 2024, while the data collection for Qwen2.5-Coder

Table 2: Model Pass@1 performance on original and multiple mutated CRUXEval and CodeNet benchmarks. Performance drops of more than 20% are highlighted.

Type	Model	Origin	FUV	AUV	AFU
CRUXEval	DeepSeek V3	65.80	45.93 (-19.87)	52.21 (-13.59)	50.63 (-15.17)
	GPT-4o mini	53.60	31.28 (**-22.32**)	42.18 (-11.42)	37.94 (-15.66)
	Llama 3.1 70B	53.20	36.79 (-16.41)	40.89 (-12.31)	37.57 (-15.63)
	Llama 3.1 8B	31.20	15.93 (-15.27)	23.26 (-7.94)	20.48 (-10.72)
	Qwen2.5-Coder 32B	65.50	37.25 (**-28.25**)	47.82 (-17.68)	39.58 (**-25.92**)
	Qwen2.5-Coder 14B	58.30	33.90 (**-24.40**)	47.71 (-10.59)	39.15 (-19.15)
	Qwen2.5-Coder 7B	45.70	31.61 (-14.09)	32.02 (-13.68)	31.96 (-13.74)
	StarCoder2 15B	33.10	12.07 (**-21.03**)	8.57 (**-24.53**)	28.36 (-4.74)
	StarCoder2 7B	33.80	20.33 (-13.47)	20.05 (-13.75)	32.49 (-1.31)
	StarCoder2 3B	30.00	16.59 (-13.41)	20.11 (-9.89)	25.19 (-4.81)
CodeNet	DeepSeek V3	94.70	78.93 (-15.77)	89.68 (-5.02)	78.54 (-16.16)
	GPT-4o mini	90.20	70.71 (-19.49)	75.48 (-14.72)	56.58 (**-33.62**)
	Llama 3.1 70B	88.00	43.21 (**-44.79**)	74.41 (-13.59)	53.17 (**-34.83**)
	Llama 3.1 8B	66.70	25.00 (**-41.70**)	30.11 (**-36.59**)	16.10 (**-50.60**)
	Qwen2.5-Coder 32B	92.70	58.21 (**-34.49**)	63.66 (**-29.04**)	51.95 (**-40.75**)
	Qwen2.5-Coder 14B	86.30	58.93 (**-27.37**)	69.89 (-16.41)	50.00 (**-36.30**)
	Qwen2.5-Coder 7B	84.10	40.71 (**-43.39**)	73.55 (-10.55)	40.24 (**-43.86**)
	StarCoder2 15B	58.60	32.86 (**-25.74**)	56.34 (-2.26)	25.12 (**-33.48**)
	StarCoder2 7B	43.00	18.21 (**-24.79**)	35.05 (-7.95)	16.34 (**-26.66**)
	StarCoder2 3B	50.70	19.29 (**-31.41**)	43.01 (-7.69)	19.27 (**-31.43**)

was completed by February 2024. Our results indicate that some models may be implicitly affected by the data contamination issue.

4.3 RQ3: Mitigation of Data Contamination

In this section, we further illustrate the significant impact of data contamination and how our method can effectively mitigate these issues. In RQ1 and RQ2, we find that the performance trends of different models under our dynamic benchmark are generally consistent. Due to computational resource constraints, in this section, we select the `Qwen2.5-Coder 1.5B Instruct` model and fine-tune it using the CRUXEval dataset to simulate the data contamination issue. In this experiment, we use a learning rate of 1e-4, a batch size of 2, and 20 training epochs. We then evaluate the model on both the original static CRUXEval benchmark and our dynamic benchmark. Since the previous evaluation shows the strong capability of multi-mutation, we use multi-mutation as the dynamic benchmarking approach. The results are shown in Figure 5.

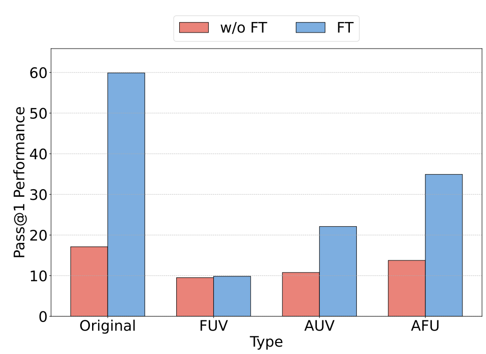

Figure 5: Comparison of Pass@1 scores on static and dynamic CRUXEval, with and without(w/o) fine-tuning the model.

It shows that the fine-tuned model achieves approximately three times higher Pass@1 scores than the original model, indicating that it has memorized the dataset. This demonstrates that data contamination can severely undermine the ability of traditional static benchmarks to accurately reflect a model's true code reasoning capabilities. Moreover, when the fine-tuned model is evaluated on the dynamic benchmark, the Pass@1 scores also decline compared to those on the static benchmark. Particularly for the FUV method, *the impact of fine-tuning on its evaluation is almost negligible*. This effectively demonstrates that our approach maintains its efficacy in assessing the model's genuine reasoning capabilities, even in scenarios where test data has been incorporated into the training dataset, thereby meeting the criteria for syntactic divergence.

4.4 RQ4: Complexity of Dynamic Benchmarks

In Table 3, we compare the similarity between benchmarks obtained by different mutated methods and original benchmarks, using the BLEU score [29] to measure the similarity. The results show that the BLEU score of the vast majority of dynamic benchmarks compared to the original benchmarks reaches *above 50*, indicating that our method effectively reduced the model's reasoning performance without significantly altering benchmarks.

Table 3: BLEU scores of different mutated benchmarks compared to the original benchmarks.

Benchmark	VI	VII	CU	F2W	CA
CRUXEval	43.35	40.70	70.41	44.45	55.64
Avatar	50.28	50.22	70.36	63.19	72.43
CodeNet	56.46	56.91	69.55	59.92	67.45
TransCoder	37.53	37.50	61.03	52.21	64.93

5 Related Work

5.1 Code Reasoning

In the domain of code, various benchmarks are developed to evaluate the code reasoning abilities of LLMs. For example, CRUXEval [9] concentrates on code execution challenges by supplying LLMs with a function and asking them either to produce outputs corresponding to given inputs or to identify a set of inputs that would yield a specified output. Meanwhile, benchmarks such as Code Lingua [28] and TransCoder [19, 33] are geared towards code translation, aiming to assess LLMs' ability to convert code from one language to another. Several other benchmarks have also been developed to assess the code reasoning capabilities of LLMs across diverse tasks. For instance, QuixBugs [23] evaluates models' proficiency in program repair, while CoDesc [12] and similar benchmarks focus on assessing models' ability to code summarization.

5.2 Mutation Testing

Mutation testing has been extensively studied as a powerful technique for evaluating the effectiveness of test suites and improving software quality. A representative example can be found in compiler testing, where mutation testing of input programs has been effectively employed to identify subtle bugs and optimization issues in modern compilers [20, 30, 22]. In the domain of LLM testing, Hooda et al. [13] conduct a comparative analysis of LLMs' performance on code generation tasks using both original and mutated datasets, revealing that current models demonstrate a limited understanding of fundamental programming concepts such as data flow and control flow. Moreover, Chen et al. [3] enhances the evaluation accuracy of large language models by modifying the meaning and context of the natural language descriptions in programming problems.

5.3 Benchmark Reliability

Benchmark reliability has been a persistent issue throughout the development of deep learning, especially since LLMs became mainstream in the academic community. The remarkable generalization capabilities and task-handling abilities of LLMs have led to the creation of numerous benchmarks tailored to various tasks. However, the quality of these benchmarks varies significantly, and prior research has already highlighted this problem. For example, Gulati et al. [10] proposed Putnam-AXIOM, demonstrating that current large models still perform poorly on mathematical problems. Similarly, Liu et al. [25] introduced EvalPlus, which rigorously evaluates LLMs' code generation capabilities through mutation-based methods. Our work further advances LLM evaluation by focusing on code reasoning tasks, thereby contributing to a more comprehensive assessment system.

6 Conclusion

In this paper, we tackle a critical challenge in model evaluation: how to keep code benchmarks meaningful when models might have already seen them during training. Our solution, dynamic benchmarking, automatically transforms test programs while keeping their semantics intact. Our experiments show this approach works remarkably well — models struggle more with our transformed benchmarks, and even when a model is fine-tuned on the original benchmark, our dynamic versions still provide reliable evaluation. This opens up new possibilities for keeping model evaluation fair and meaningful as models continue to advance.

References

[1] Rachith Aiyappa, Jisun An, Haewoon Kwak, and Yong-Yeol Ahn. Can we trust the evaluation on chatgpt? *arXiv preprint arXiv:2303.12767*, 2023.

[2] 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, 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, Josh 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, 2021. URL `https://arxiv.org/abs/2107.03374`.

[3] Simin Chen, Pranav Pusarla, and Baishakhi Ray. Dynamic benchmarking of reasoning capabilities in code large language models under data contamination, 2025. URL `https://arxiv.org/abs/2503.04149`.

[4] Cursor. Cursor: The ai-first code editor, 2023. URL `https://www.cursor.com/`. Accessed: 2023-10-30.

[5] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, et al. Deepseek-v3 technical report, 2024. URL `https://arxiv.org/abs/2412.19437`.

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[7] Nat Friedman. Introducing github copilot: your ai pair programmer. *https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer*, 2021.

[8] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, et al. The llama 3 herd of models, 2024. URL `https://arxiv.org/abs/2407.21783`.

[9] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

[10] Aryan Gulati, Brando Miranda, Eric Chen, Emily Xia, Kai Fronsdal, Bruno de Moraes Dumont, and Sanmi Koyejo. Putnam-axiom: A functional and static benchmark for measuring higher level mathematical reasoning. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

[11] Valentin Hartmann, Anshuman Suri, Vincent Bindschaedler, David Evans, Shruti Tople, and Robert West. Sok: Memorization in general-purpose large language models, 2023. URL `https://arxiv.org/abs/2310.18362`.

[12] Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md. Mahim Anjum Haque, Tahmid Hasan, Wasi Ahmad, Anindya Iqbal, and Rifat Shahriyar. CoDesc: A large code–description parallel dataset. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 210–218, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.18. URL `https://aclanthology.org/2021.findings-acl.18/`.

[13] Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Do large code models understand programming concepts? a black-box approach. *arXiv preprint arXiv:2402.05980*, 2024.

[14] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[15] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL `https://arxiv.org/abs/2409.12186`.

[16] Yoichi Ishibashi and Yoshimasa Nishimura. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization, 2024. URL `https://arxiv.org/abs/2404.02183`.

[17] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL `https://arxiv.org/abs/2406.00515`.

[18] Ahmed Khanfir, Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Codebert-nt: code naturalness via codebert, 2022. URL `https://arxiv.org/abs/2208.06042`.

[19] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020. URL `https://arxiv.org/abs/2006.03511`.

[20] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[21] Kyla Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. Chatdbg: An ai-powered debugging assistant, 2024. URL `https://arxiv.org/abs/2403.16354`.

[22] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. Boosting compiler testing by injecting real-world code. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656386. URL `https://doi.org/10.1145/3656386`.

[23] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[24] Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. Codemind: A framework to challenge large language models for code reasoning, 2024. URL `https://arxiv.org/abs/2402.09664`.

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[26] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024. URL `https://arxiv.org/abs/2402.19173`.

[27] OpenAI, Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[28] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 866–866. IEEE Computer Society, 2024.

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[30] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, pages 849–863, 2016.

[31] Niklaus Wirth, Niklaus Wirth, Niklaus Wirth, Suisse Informaticien, and Niklaus Wirth. *Compiler construction*, volume 1. Addison-Wesley Reading, 1996.

[32] Chen Yang, Junjie Chen, Jiajun Jiang, and Yuliang Huang. Dependency-aware code naturalness. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi: 10.1145/3689794. URL `https://doi.org/10.1145/3689794`.

[33] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation, 2024. URL `https://arxiv.org/abs/2404.14646`.

A Benchmark Test Case Counts

Table 4 enumerates the number of test cases in each benchmark and mutation method. Note that not all test cases can be mutated by each method. For example, out of all 800 test cases in CRUXEval, there are 306 cases containing *for* loops and thus can be mutated by *For2While*. In our evaluation, we only count the mutated test cases when evaluating each mutation method.

Table 4: #test cases in the original benchmarks and #test cases that can be mutated by each mutation method.

Mutation	CRUXEval	Avatar	CodeNet	TransCoder
Original	800	250	200	568
VarNormI	785	131	150	568
VarNormII	785	131	150	568
ConstUnfold	455	239	169	546
For2While	306	181	100	349
CondAug	374	191	134	396

B Detailed Experimental Results

In this section, we present all of our experimental results, which generally align with the trends reported in Section 4.

B.1 More Results on Single Mutated Experiments

More experimental results on single mutated benchmark are displayed in Table 5.

B.2 More Results on Multiple Mutated Experiments

More experimental results on single mutated benchmark are displayed in Table 6.

Table 5: Model Pass@1 performance on original and single mutated Avatar and TransCoder benchmarks. Performance drops of more than 10% are highlighted.

Type	Model	Origin	VarNormI	VarNormII	ConstUnfold	For2While	CondAug
Avatar	DeepSeek V3	80.64	85.42 (+4.78)	81.98 (+1.34)	72.68 (-7.96)	73.48 (-7.16)	81.68 (+1.04)
	GPT-4o mini	72.48	82.13 (+9.65)	74.05 (+1.57)	54.77 (**-17.71**)	67.90 (-4.58)	73.25 (+0.77)
	Llama 3.1 70B	72.80	64.35 (**-8.45**)	72.6 (-0.20)	57.87 (**-14.93**)	62.71 (**-10.09**)	70.16 (-2.64)
	Llama 3.1 8B	46.48	49.01 (+2.53)	52.98 (+6.50)	29.87 (**-16.61**)	36.24 (**-10.24**)	43.04 (-3.44)
	Qwen2.5 32B	75.64	78.76 (+3.12)	76.64 (+1.00)	59.42 (**-16.22**)	65.08 (**-10.56**)	70.68 (-4.96)
	Qwen2.5 14B	65.20	70.08 (+4.88)	70.53 (+5.33)	54.31 (**-10.89**)	57.46 (-7.74)	64.08 (-1.12)
	Qwen2.5 7B	64.60	61.83 (-2.77)	65.80 (+1.20)	43.18 (**-21.42**)	51.27 (**-13.33**)	58.12 (-6.48)
	StarCoder2 15B	58.12	43.66 (**-14.46**)	45.95 (**-12.17**)	25.86 (**-32.26**)	26.52 (**-31.60**)	30.68 (**-27.44**)
	StarCoder2 7B	40.00	30.84 (-9.16)	28.85 (**-11.15**)	16.40 (**-23.60**)	20.88 (**-19.12**)	27.54 (**-12.46**)
	StarCoder2 3B	41.80	32.06 (-9.74)	32.37 (-9.43)	17.49 (**-24.31**)	17.90 (**-23.90**)	23.14 (**-18.66**)
TransCoder	Deepseek V3	79.36	75.84 (-3.52)	74.69 (-4.67)	69.73 (-9.63)	65.39 (**-13.97**)	67.95 (**-11.41**)
	GPT-4o mini	78.11	72.28 (-5.83)	71.82 (-6.29)	69.56 (-8.55)	82.96 (+4.85)	70.95 (-7.16)
	Llama 3.1 70B	78.88	73.36 (-5.52)	72.57 (-6.31)	52.69 (**-26.19**)	61.55 (**-17.33**)	64.82 (**-14.06**)
	Llama 3.1 8B	72.56	62.66 (-9.90)	64.23 (-8.33)	59.04 (**-13.52**)	77.80 (+5.24)	64.55 (-8.01)
	Qwen2.5 32B	73.24	55.48 (**-17.76**)	67.88 (-5.36)	71.34 (-1.90)	75.27 (+2.03)	70.89 (-2.35)
	Qwen2.5 14B	78.88	71.45 (-7.43)	71.08 (-7.80)	74.11 (-4.77)	77.95 (-0.93)	71.26 (-7.62)
	Qwen2.5 7B	67.87	65.60 (-2.27)	57.47 (**-10.40**)	58.01 (-9.86)	62.76 (-5.11)	57.66 (**-10.21**)
	StarCoder2 15B	51.33	49.05 (-2.28)	42.07 (-9.26)	38.40 (**-12.93**)	29.68 (**-21.65**)	34.89 (**-16.44**)
	StarCoder2 7B	54.71	40.08 (**-14.63**)	45.31 (-9.40)	33.90 (**-20.81**)	22.32 (**-32.39**)	34.52 (**-20.19**)
	StarCoder2 3B	48.15	36.31 (**-11.84**)	34.56 (**-13.59**)	19.91 (**-28.24**)	10.74 (**-37.41**)	18.28 (**-29.87**)

Table 6: Model Pass@1 performance on original and multiple mutated Avatar and TransCoder benchmarks. Performance drops of more than 20% are highlighted.

Type	Model	Origin	FUV	AUV	AFU
Avatar	DeepSeek V3	80.64	70.24 (-10.40)	74.75 (-5.89)	64.39 (-16.25)
	GPT-4o mini	72.48	51.95 (**-20.53**)	61.62 (-10.86)	41.16 (**-31.32**)
	Llama 3.1 70B	72.80	48.29 (**-24.51**)	54.34 (-18.46)	42.58 (**-30.22**)
	Llama 3.1 8B	46.48	20.73 (**-25.75**)	18.79 (**-27.69**)	12.65 (**-33.83**)
	Qwen2.5-Coder 32B	75.64	57.32 (**-18.32**)	52.32 (**-23.32**)	39.48 (**-36.16**)
	Qwen2.5-Coder 14B	65.20	49.02 (-16.18)	46.06 (**-19.14**)	36.39 (**-28.81**)
	Qwen2.5-Coder 7B	64.60	34.88 (**-29.72**)	41.01 (**-23.59**)	24.13 (**-40.47**)
	StarCoder2 15B	58.12	19.76 (**-38.36**)	25.25 (**-32.87**)	20.16 (**-37.96**)
	StarCoder2 7B	40.00	14.39 (**-25.61**)	15.15 (**-24.85**)	12.00 (**-28.00**)
	StarCoder2 3B	41.80	10.00 (**-31.80**)	16.57 (**-25.23**)	8.00 (**-33.80**)
TransCoder	DeepSeek V3	79.36	62.47 (-16.89)	66.22 (-13.79)	72.91 (-6.45)
	GPT-4o mini	78.11	45.16 (**-32.95**)	49.05 (**-29.06**)	57.09 (**-21.44**)
	Llama 3.1 70B	78.88	51.94 (**-26.94**)	58.34 (**-20.54**)	59.27 (**-24.89**)
	Llama 3.1 8B	72.56	25.16 (**-47.40**)	31.51 (**-41.05**)	22.82 (**-49.74**)
	Qwen2.5-Coder 32B	73.24	56.25 (-17.00)	54.58 (**-25.48**)	58.55 (**-20.04**)
	Qwen2.5-Coder 14B	78.88	52.36 (**-26.52**)	57.72 (**-21.16**)	58.55 (**-20.33**)
	Qwen2.5-Coder 7B	67.87	27.30 (**-40.57**)	42.75 (**-25.12**)	45.98 (**-21.89**)
	StarCoder2 15B	51.33	12.79 (**-38.54**)	29.97 (**-21.36**)	23.64 (**-27.69**)
	StarCoder2 7B	54.71	7.56 (**-47.15**)	8.92 (**-45.79**)	11.27 (**-43.44**)
	StarCoder2 3B	48.15	3.32 (**-44.83**)	9.41 (**-38.74**)	13.84 (**-34.31**)

C Prompts

In this section, we present the prompts we use to instruct the model to do code execution tasks and code translation tasks in our experiments.

> Based on the given Python code, which may contain errors, complete the assert statement with the output when executing the code on the given test case. Do NOT output any extra information, even if the function is incorrect or incomplete. Output "# done" after the assertion.

> You are a code translation expert. Translate the Python code below to Java. Do NOT output any extra information.

D Case Study

In this section, we present some cases generated in our experiments in Figure 6 and Figure 7.

Figure 6 shows two examples of code execution tasks performed by the GPT-4o mini and Qwen2.5-Coder 32B models. In Figure 6(a), the LLM was originally able to make the correct judgment, but after applying the Constant Unfolding mutation, it made an incorrect judgment. Figure 6(b) presents a slightly more complex example. After the FUV mutation, the model incorrectly evaluated the `isalpha` condition and output an erroneous result.

Figure 7 shows two examples of code translation tasks performed by the DeepSeek V3 and Llama 3.1 8B models. In Figure 7(a), the key to correct translation lies in paying attention to the sign of the numbers in the array. Since Python and Java define the modulo operation for negative numbers differently, this detail must be carefully considered when translating from Python to Java. In the original code, the LLM successfully recognized this aspect. However, after applying the Constant Unfolding mutation, the LLM overlooked this detail and ultimately produced code that could not pass the test. Additionally, since Python does not explicitly specify types, correctly determining the type is also a crucial aspect of translating from Python to Java. In Figure 7(b), the original code successfully identifies the most suitable type for the code. However, after the FUV mutation, the model makes an incorrect judgment.

```python
def f(x):
    n = len(x)
    i = 0
    while i < n and x[i].isdigit():
        i += 1
    return i == n
assert f('1') == True
```

```python
def f(x):
    n = len(x)
    i = (3 + -3)
    while i < n and x[i].isdigit():
        i += (66 + -65)
    return i == n
assert f('1') == False
```

(a) CRUXEval example generated by GPT-4o mini.

```python
def f(doc):
    for x in doc:
        if x.isalpha():
            return x.capitalize()
    return '-'
assert f('raruwa') == 'R'
```

```python
def iXN(aRL):
    jLn = (25 + -25)
    Fbd = aRL
    while jLn < len(Fbd):
        XBu = Fbd[jLn]
        if XBu.isalpha():
            return XBu.capitalize()
        jLn = jLn + (21 + -20)
    return '-'
assert iXN('raruwa') == '-'
```

(b) CRUXEval example generated by Qwen2.5-Coder 32B

Figure 6: Case studies of code execution.

```
A, B, C, D = map(int, input().split())
m = 998244353
d = [[0] * (D + 1) for _ in range(C + 1)]
d[A][B] = 1
for i in range(A, C + 1):
    for j in range(B, D + 1):
        d[i][j] += (d[i][j - 1] * i + d[i - 1][j] * j -
                    d[i - 1][j - 1] * (i - 1) * (j - 1)) %
m
print(d[C][D])
```

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int A = scanner.nextInt();
        int B = scanner.nextInt();
        int C = scanner.nextInt();
        int D = scanner.nextInt();
        int m = 998244353;
        long[][] d = new long[C + 1][D + 1];
        d[A][B] = 1;
        for (int i = A; i <= C; i++) {
            for (int j = B; j <= D; j++) {
                if (i == A && j == B) continue;
                d[i][j] = (d[i][j - 1] * i + d[i - 1][j] *
j - d[i - 1][j - 1] * (i - 1) * (j - 1)) % m;
                if (d[i][j] < 0) d[i][j] += m;
            }
        }
        System.out.println(d[C][D]);
    }
}
```

```
A, B, C, D = map(int, input().split())
m = (79 + 998244274)
d = [[(19 + -19)] * (D + (20 + -19)) for _ in range(C +
(57 + -56))]
d[A][B] = (2 + -1)
for i in range(A, C + (99 + -98)):
    for j in range(B, D + (48 + -47)):
        d[i][j] += (d[i][j - (44 + -43)] * i + d[i - (25 +
-24)][j] * j -
                    d[i - (76 + -75)][j - (6 + -5)] * (i -
(35 + -34)) * (j - (15 + -14))) % m
print(d[C][D])
```

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int A = scanner.nextInt();
        int B = scanner.nextInt();
        int C = scanner.nextInt();
        int D = scanner.nextInt();
        int m = 79 + 998244274;
        int[][] d = new int[C + 1][D + 1];
        d[A][B] = 1;
        for (int i = A; i <= C; i++) {
            for (int j = B; j <= D; j++) {
                if (i == A && j == B) continue;
                d[i][j] = (d[i][j - 1] * i + d[i - 1][j] *
j - d[i - 1][j - 1] * (i - 1) * (j - 1)) % m;
            }
        }
        System.out.println(d[C][D]);
    }
}
```

(a) Avatar example generated by DeepSeek V3.

```
def repeat ( s , n ) :
    s1 = s
    for i in range ( 1 , n ) :
        s += s1
    return s
```

```
public static String f_filled(String s, int n) {
    String s1 = s;
    for (int i = 1; i < n; i++) {
        s += s1;
    }
    return s;
}
```

```
def VoU ( api , kDU ) :
    AFe = api
    cWi = (91 + -91)
    bYk = range ( (42 + -41) , kDU )
    while cWi < len(bYk):
        qQd = bYk[cWi]
        api += AFe
        cWi = cWi + (64 + -63)
    return api
```

```
public static int f_filled(int[] api, int kDU) {
    int AFe = api[0];
    int cWi = (91 + -91);
    int[] bYk = new int[kDU];
    for (int i = (42 + -41); i < kDU; i++) {
        bYk[i] = i;
    }
    while (cWi < kDU) {
        int qQd = bYk[cWi];
        api[0] += AFe;
        cWi = cWi + (64 + -63);
    }
    return api[0];
}
```

(b) TransCoder example generated by Llama 3.1 8B.

Figure 7: Case studies of code translation.

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: Abstract and Section 1.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: Section 4.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

   Answer: [NA]

Justification: The paper does not include theoretical results.

Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: Section 3, Section 4 and Appendix.

   Guidelines:

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

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

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

Answer: [Yes]

Justification: We show it in OpenReview.

Guidelines:

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: Section 3, Section 4 and Appendix.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [No]

   Justification: Error bars are not reported because it would be too computationally expensive.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: Section 3.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: All the paper.

   Guidelines:

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

10. **Broader impacts**

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

    Answer: [NA]

    Justification: This paper primarily examines issues related to benchmarks and does not have a direct connection to social impact.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

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

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: Section 3.

    Guidelines:

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

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

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

    Answer: [Yes]

    Justification: We show it in OpenReview.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

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

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

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

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

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

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

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

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

16. **Declaration of LLM usage**

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

Answer: [Yes]

Justification: Section 3.

Guidelines:

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