Memorize or Generalize? Evaluating LLM Code Generation with Evolved Questions

Anonymous ACL submission

Abstract

Large Language Models (LLMs) are known to exhibit a *memorization*[1] phenomenon in code generation: instead of truly understanding the underlying principles of a programming problem, they tend to memorize the original prompt and its solution together in the training. Consequently, when facing variants of the original problem, their answers very likely resemble the memorized solutions and fail to generalize. In this paper, we investigate this phenomenon by designing three evolution strategies to create variants: *mutation*, *paraphrasing*, and *code-rewriting*. By comparing the performance and AST similarity of the LLM-generated codes before and after these three evolutions, we develop a memorization score that positively correlates with the level of memorization. As expected, as supervised fine-tuning goes on, the memorization score rises before overfitting, suggesting more severe memorization. We demonstrate that common mitigation approaches, such as prompt translation and using evolved variants as data augmentation in supervised learning and reinforcement learning, either compromise the performance or fail to alleviate the memorization issue. Therefore, memorization remains a significant challenge in LLM code generation, highlighting the need for a more effective solution.

1 Introduction

Large Language Models (LLMs) have shown remarkable versatility in tasks ranging from text generation and question answering to automated code generation. In the coding domain, models such as Qwen-Coder (Hui et al., 2024), CodeLlama (Rozière et al., 2024), and DeepSeek-Coder (Guo et al., 2024) have pushed the boundaries of translating natural language into code. However, these models often rely on recalling previously seen training examples rather than genuinely reasoning about new problems (Xia et al., 2024). This over-reliance on memorization can weaken a model's ability to adapt to novel or slightly varied coding tasks.

Recent studies have highlighted this issue through evaluation benchmarks: when minor alternations are made to a coding prompt, LLM performance can drop significantly, suggesting limited generalization (Xia et al., 2024). Evolved benchmarks such as EvalPlus (Liu et al., 2023) and Evo-Eval (Xia et al., 2024) thus aim to reduce memorization biases by introducing subtle modifications to existing tasks. Despite these efforts, the phenomenon of memorization in code LLMs remains insufficiently understood. In particular, it is difficult to distinguish true problem-solving ability from simple rote recall of training data.

Although prior work has explored memorization in code (Yang et al., 2024), existing definitions based on natural language (Carlini et al., 2023) may not transfer to code. Unlike typical text, code's syntax is as crucial as its semantics, motivating the use of Abstract Syntax Trees (ASTs) to capture structural similarity. Moreover, researchers frequently evaluate code generation using functional correctness metrics such as pass@k (Lyu et al., 2024), indicating that memorization should be assessed with both structural and functional criteria.

One might interpret memorization as a form of overfitting, but we distinguish between *early-stage* and *late-stage* memorization in Figure 3. Late-stage memorization aligns closely with traditional overfitting—where performance degrades on validation sets while the model repeats patterns learned from the training set. Early-stage memorization, however, arises even when the model is not yet overfitted, yet still reproduces code patterns from seen examples. Our focus in this work is on *early-stage memorization*, and any further mention of "memorization" refers to this phase.

To investigate this phenomenon, we mutate ex-

[1] In this paper, we focus on memorization before overfitting because late-stage memorization resembles overftting.

isting coding tasks with three evolution methods: *mutation*, *paraphrasing*, and *code-rewriting*. These methods correspond to adding noise in text space, semantic space and code space respectively. Therefore, we refer to this as multi-level evolution 1. Using MBPP-Plus (Liu et al., 2023) as our base, we construct three altered datasets and observe the performance on these evolved tasks.

We further propose a *memorization score* to quantify memorization. This score combines (i) functional correctness (code accuracy) and (ii) structural overlap (AST similarity). In our experiments, the memorization score correlates positively with the extent of memorization, increasing as fine-tuning proceeds and helping differentiate early-stage memorization across different models. Finally, we explore three mitigation strategies—*supervised fine-tuning*, *reinforcement learning*, and *problem translation*—to reduce memorization scores. While each method partially mitigate memorization, we observe a trade-off: mitigating memorization can also degrade performance on the original tasks. This leaves open the challenge of designing training methods that limit rote repetition without sacrificing overall capabilities.

Contributions. In summary, our work:

- Introduces a **multi-level evolution framework** (mutation, paraphrasing, code-rewriting) to evaluate how LLMs handle evolved coding tasks.
- Proposes a **memorization score** that integrates both functional correctness and AST-based structural similarity to measure early-stage memorization in code LLMs.
- Demonstrates that code-specialized LLMs exhibit high memorization behaviors, and that **three mitigation methods** reduce memorization but often at the cost of performance on the original dataset.

We hope these findings could offer insights into why LLMs experience performance drops when presented with minor task mutations and provide actionable strategies to quantify and mitigate memorization in code generation.

2 Related Work

2.1 Memorization

Memorization refers to the ability of neural network models to memorize and reproduce their training data (Carlini et al., 2019),(Bayat et al., 2024). In the era of large language models (LLMs), researchers have proposed several new definitions of memorization specific to LLMs (Carlini et al., 2023), (Zhang et al., 2023), (Schwarzschild et al., 2024). Beyond natural language, studies on memorization in logical reasoning (Xie et al., 2024) and code generation (Yang et al., 2024) further underscore the increasing importance of understanding memorization in LLMs.

2.2 Data Synthesis in Code Generation

To enhance the quality of training data in LLM code generation (Jiang et al., 2024), researchers have developed several human-generated datasets, such as Humaneval (Chen et al., 2021), MBPP (Austin et al., 2021) and so on. However, due to the high cost and limited quantity of handcrafted datasets, data synthesis has emerged as a viable solution, which can be categorized into Self-Instruct (Wang et al., 2023), Evol-Instruct (Luo et al., 2023) and OSS-Instruct (Wei et al., 2024). Furthermore, the evolution idea can be generalized into other code-like domains, such as mathematical problems (Gulati et al., 2024) and logical reasoning (Xie et al., 2024), suggesting its significant effectiveness.

3 Methodology

3.1 Multi-level Evolution

We denote a professional model that can always get the ground truth of a programming problem as G, the embedding projection layer as E, the text space as T, the semantic space of natural language as S, and the code space as C.

Then given a coding problem text as x, the corresponding embedding vector is $y = E(x)$ and the ground truth solution is $z = G[E(x)] = G(y)$. Therefore, we can formalize the code generation process as a mapping from natural language to code language:

$$y = E(x) : T \mapsto S$$
$$z = G(y) : S \mapsto C$$

Figure 1 illustrates our evolution methods. We begin by distinguishing between two scenarios: one in which the ground truth code for the original problem remains unchanged and one in which it does not. Specifically, mutation evolution and paraphrasing evolution preserve the original ground truth, whereas code-rewriting evolution generates a new ground truth.

For mutation evolution, we simply ask LLM to do word scrambling, text shift and random capitalization to the original prompt. Then it is actually

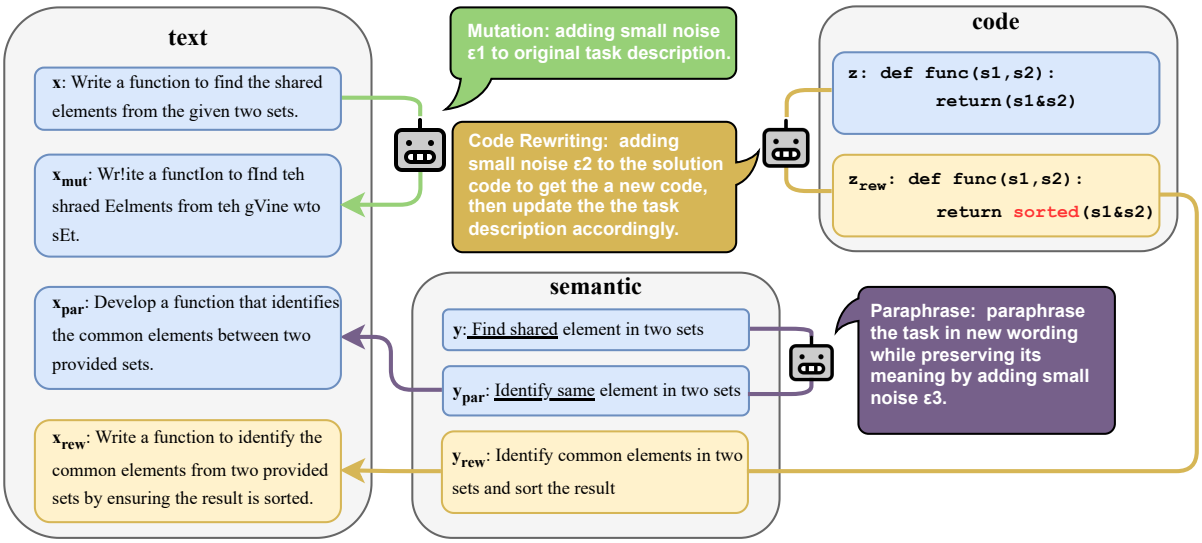

Figure 1: The workflow of the multi-level evolution methods in text, semantic, and code spaces. Boxes inside those spaces with the same color share the same canonical solutions. Mutation (mut), paraphrasing (par) and code-rewriting (rew) add noise in text space, semantic space and code space respectively. And finally they will be mapped back to text space as evolved problem $x_{\text{mut}}, x_{\text{par}}, x_{\text{rew}}$. The evolution process of adding noise and mapping are all conducted by the professional model G (GPT-4o), illustrated as the gray robot. We denote this framework as multi-level evolution.

adding a small noise ϵ_1 to original problem text:

$$x_{mut} = x + \epsilon_1, \quad \epsilon_1 \in T$$
$$s.t. \quad G[E(x_{mut})] = G[E(x)]$$

For paraphrasing evolution, we ask the model to paraphrase the original prompt in fresh wording or sentence structure. The goal is to keep the same meaning but change how it's phrased—adding a small noise ϵ_2 in the semantic space of natural language S:

$$x_{par} = E^{-1}[E(x) + \epsilon_2], \quad \epsilon_2 \in S$$
$$s.t. \quad G[E(x_{par})] = G[E(x)]$$

Additionally, we also suppose a professional model G that can summarize a prompt from a given code, then we have:

$$y = G^{-1}(z) : C \mapsto S$$

For the code-rewriting evolution, we first feed the original code solution into the professional LLM (like GPT-4o), asking it to do some slight modification. Noticed that the modification is not simple variable renaming or constant change, but logical and structural difference from origin code. After that we ask the LLM to change the origin problem according to the new code solution while maintaining the least editing distance. Therefore, code-rewriting can be regarded as adding a small noise ϵ_3 in the code space C, and the ground truth won't be the same:

$$x_{rew} = E^{-1}[G^{-1}[G[E(x)] + \epsilon_3]], \quad \epsilon_3 \in C$$
$$s.t. \quad G[E(x_{rew})] \neq G[E(x)]$$

The three evolution methods progressively increase in complexity, introducing noise in the text space, semantic space, and code space, respectively. Unlike previous code evolution methods, these varying levels of evolution aim to provide deeper insights into an LLM's true capabilities. Hence, we refer to this framework as multi-level evolution 1.

3.2 Evaluation Framework

3.2.1 Overall Accuracy

Accuracy is defined as the proportion of programming tasks for which the model-generated code passed all associated test cases. Specifically, for each task, if the solution correctly produces the expected output for every test case, it is considered a successful pass. The **overall accuracy** is then calculated as the number of successfully passed tasks divided by the total number of tasks evaluated.

Formally, let $\mathcal{T}$ denote the set of all programming tasks. For each task $i \in \mathcal{T}$, define an indicator

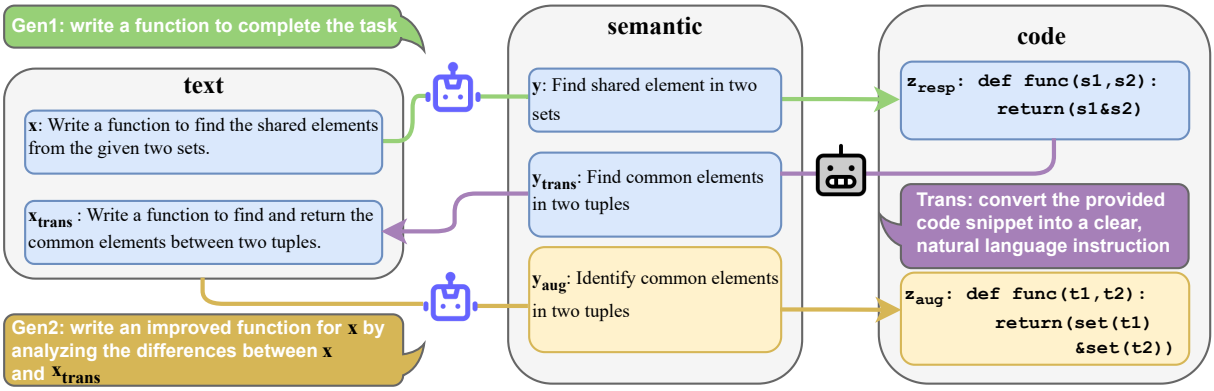

Figure 2: The figure of the problem translation process. The same color of inner boxes inside each space share the same canonical solutions. We first ask target model M (the blue robot) to generate a code response z_{resp} based on x (Gen1), then use professional model G (the gray robot) to translate it back into a new code x_{trans} (Trans); finally, we ask the tar model G to figure out their difference and generating the final response z_{aug}.

function:

$$I_i = \begin{cases} 1, & \text{if code passes all tests for task } i, \\ 0, & \text{otherwise.} \end{cases}$$

Then, the overall accuracy $Acc(\mathcal{T})$ is defined as:

$$Acc(\mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} I_i.$$

3.2.2 Overall AST Similarity

For each programming task, we obtain the similarity score between the canonical solution and the candidate response by using an AST-Based Source Code Similarity Detection Tool (AnubisLMS, 2023). Let S_i be the AST similarity score between the **candidate response** of task i in set $\mathcal{T}_1$ and the **canonical solution** of that in set $\mathcal{T}_2$, and the overall similarity is given by:

$$sim_{\mathcal{T}_2}(\mathcal{T}_1) = \frac{1}{|\mathcal{T}_1|} \sum_{i \in |\mathcal{T}_1|} S_i.$$

For clarity, we assume that task i in $\mathcal{T}_1$ corresponds to task i in $\mathcal{T}_2$, ensuring a one-to-one pairing between the tasks in the two sets.

3.2.3 Memorization Score

We proposed a memorization score to quantify memorization. Given the set of coding problems $\mathcal{T}$, We denote its mutation, paraphrasing and code-rewriting problem dataset as $\mathcal{T}_{\text{mut}}, \mathcal{T}_{\text{par}}, \mathcal{T}_{\text{rew}}$ respectively. Then for a set of given problems $\mathcal{T}$, the memorization score can be calculated as below:

$$\begin{aligned} \text{Mem}(\mathcal{T}) = & \frac{1}{3} * [(Acc(\mathcal{T}) - Acc(\mathcal{T}_{\text{mut}})) + \\ & (Acc(\mathcal{T}) - Acc(\mathcal{T}_{\text{par}})) + \\ & (Acc(\mathcal{T}) - Acc(\mathcal{T}_{\text{rew}}))] \\ & + \max(0, sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}}) - sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}}))) \end{aligned}$$

The score ranges from 0 to 2, with higher values indicating more severe memorization. We will explain it from two aspects:

The first three items measure the accuracy drop between original and evolved problems. An increased gap indicates high LLM performance on original problems $\mathcal{T}$ but poor performance on evolved ones, suggesting a lack of true reasoning and increased reliance on memorization. We use the factor $\frac{1}{3}$ for normalization.

The last item compares two AST similarities: between the code response and the canonical solution on the code-rewriting dataset, and between that response and the original dataset's canonical solution. A negative difference indicates that the code response on the code-rewriting dataset closely matches its canonical solution, implying no contribution to memorization, so the value is set to 0. Conversely, a positive difference suggests that the response aligns more with the original dataset's solution, indicating reliance on the original data. A higher value indicates that the LLM is relying more on the original data to get the answer.

Thus, the accuracy gap highlights memorization from a functionality perspective, while the AST similarity gap does so from a structural angle. High

values in both suggest severe memorization issues. We combine these normalized metrics to create a memorization score, quantifying memorization within LLMs during code generation.

3.3 Mitigation Methods

3.3.1 Supervised Fine-Tuning

Supervised Fine-Tuning adapts a pre-trained model to a specific task by training it on a labeled dataset, teaching it to predict the correct label for each input. In our setup, coding problems are the inputs, and code solutions are the labels. We use two dataset combinations: a code-rewriting dataset and a half-and-half origin-rewriting dataset, to study the impact of varying exposure to the origin dataset. Additionally, we also examine early-stage memorization, selecting the epoch before overfitting to assess our mitigation methods.

3.3.2 Reinforcement Learning

In the large language model era, Reinforcement Learning enhances fine-tuning efficiency. A leading method is Direct Preference Optimization (DPO)(Rafailov et al., 2024), which optimizes policy based on preferences without a defined reward model, using a simple classification objective. We use the same reference and baseline models to constrain parameters, designating the code solution from the code-rewriting dataset as the winner label, and the original training dataset's solution as the loser label.

3.3.3 Problem Translation

The workflow of problem translation is shown in Figure 2. We first ask the target model M to generate the code response z for problem x:

$$z = M(y) = M[E(x)]$$

Then we ask the professional model G to translate the response $\hat{z}$ back to a coding problem x':

$$x_{\text{trans}} = E^{-1}[G^{-1}(z)]$$

After that we have two problems: the origin problem x and translated problem x'. Now we ask the target model M to distinguish the differences between them, and then generate the final code response z' again based on the origin problem and the difference. $\oplus$ means concatenating prompts from two sides.

$$z_{\text{aug}} = M[E(x \oplus \text{diff}(x, x_{\text{trans}}))]$$

Because this process includes the inverse mapping from code space back to text space, we aim for the target model can learn the mapping bias between text space and code space from the translated problem. This understanding is expected to enhance performance on the code-rewriting dataset and ultimately diminish the memorization phenomenon.

Through these three mitigation methods, we aim to alleviate the performance gap between the original and evolved datasets while reducing memorization by enhancing the model's reasoning and generalization abilities.

4 Experiments

4.1 Dataset

We use the MBPP-Plus (Liu et al., 2023) as our origin dataset. Rewritten prompts and solutions can alter input formats, making them incompatible with original test cases due to mismatched input types rather than incorrect logic. To ensure integrity, we filtered out such tasks, retaining **283** tasks. To distinguish the early-stage memorization from overfitting, we split the dataset into a 4:1 train-valid ratio, resulting in 226 training and 57 validation items. For simplicity, we denote the training and validation sets as $\mathcal{T}$ and $\mathcal{T}_{\text{valid}}$, respectively. Then we adopt the evolution methods described in Section 3.1 and curate three evolved datasets, code-rewriting ($\mathcal{T}_{rew}$), paraphrasing ($\mathcal{T}_{par}$), and mutation ($\mathcal{T}_{mut}$), based on the original training set to explore code memorization in LLMs. Note that the validation set only detects overfitting, and the other three evolved datasets are based on the training set. More details about the MBPP-Plus and our evolved datasets can be found in the appendix B.

4.2 Result Analysis

4.2.1 Memorization Exists in LLM Code Generation

To illustrate the presence of memorization effects in LLM code generation, we compare baseline models with their instruction-tuned counterparts: Llama-3.1-8B versus Llama-3.1-8B-Instruct, and Qwen2.5-7B versus Qwen2.5-Coder-7B, as shown on Table 1. More details about these models can be found in the appendix C. On all four datasets, Llama-3.1-8B-Instruct substantially outperforms its baseline version, showing an approximate 25% increase in accuracy. In contrast, Qwen2.5-Coder-7B shows large increases around

Model	$Acc(\mathcal{T})$	$Acc(\mathcal{T}_{\text{rew}})$	$Acc(\mathcal{T}_{\text{mut}})$	$Acc(\mathcal{T}_{\text{par}})$	$sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$	$sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$	Mem($\mathcal{T}$)
Llama-3.1-8B	0.173	0.133	0.159	0.189	0.132	0.151	0.013
Llama-3.1-8B-Instruct	0.407	0.420	0.327	0.425	0.133	0.169	0.016
Qwen2.5-7B	0.465	0.385	0.416	0.469	0.149	0.205	0.041
Qwen2.5-Coder-7B	0.615	0.442	0.473	0.628	0.157	0.230	0.100

Table 1: Evaluation of baseline models on the train set variants. We report accuracy on the original train set ($Acc(\mathcal{T})$) and three variants: code-rewriting ($Acc(\mathcal{T}_{\text{rew}})$), mutation ($Acc(\mathcal{T}_{\text{mut}})$), and paraphrasing ($Acc(\mathcal{T}_{\text{par}})$). Similarity scores between original and rewritten sets ($sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$) and within the rewritten set ($sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$) are provided, along with a memorization score on the original set (Mem($\mathcal{T}$)). The highest memorization score is highlighted in red.

Epoch	$Acc(\mathcal{T})$	$Acc(\mathcal{T}_{\text{rew}})$	$Acc(\mathcal{T}_{\text{mut}})$	$Acc(\mathcal{T}_{\text{par}})$	$Acc(\mathcal{T}_{\text{valid}})$	$sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$	$sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$	Mem($\mathcal{T}$)
baseline	0.615	0.442	0.473	0.628	0.561	0.157	0.230	0.100
10	0.654	0.367	0.566	0.65	0.561	0.295	0.217	0.200
20	0.823	0.429	0.739	0.814	0.474	0.499	0.234	0.428
30	0.863	0.398	0.805	0.845	0.544	0.571	0.277	0.474
40	0.881	0.416	0.801	0.867	0.509	0.56	0.277	0.469
50	0.938	0.429	0.836	0.907	0.474	0.605	0.275	0.544
60	0.942	0.429	0.841	0.925	0.456	0.612	0.279	0.544

Table 2: Evaluation of Qwen2.5-Coder-7B during supervised fine-tuning. Each row shows performance across epochs. We report accuracy on the train set ($Acc(\mathcal{T})$), its code-rewriting ($Acc(\mathcal{T}_{\text{rew}})$), mutation ($Acc(\mathcal{T}_{\text{mut}})$), paraphrasing ($Acc(\mathcal{T}_{\text{par}})$) variants, and the validation set ($Acc(\mathcal{T}_{\text{valid}})$). Also included are similarity scores ($sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$, $sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$) and a memorization score (Mem($\mathcal{T}$)). The red-highlighted epoch marks the onset of overfitting. Full results are in Table 5 in the appendix.

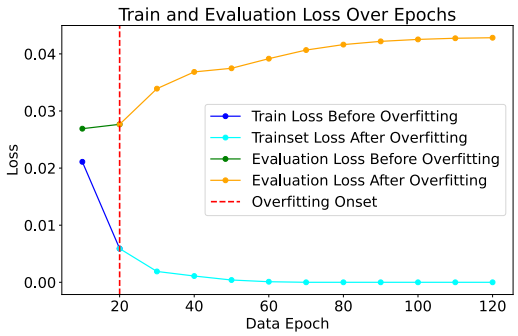

Figure 3: The loss curve of Qwen2.5-Coder-7B when fine-tuned on the train dataset. We can find that the evaluation loss begins to increase significantly at epoch 20 (red line), which stands for the LLM starts to overfit on the train dataset. Memorization then can be divided into early-stage (before red line) and late-stage (after red line) memorization. Considering the similar effects between late-stage memorization and overfitting, we explore the early-stage memorization before overfitting.

15% primarily on the origin and paraphrasing sets, while improvements on the mutation and code-rewriting sets are comparatively lower (around 5%). We hypothesize that the instruction-tuning process in Qwen2.5-Coder-7B may have introduced unanticipated memorization effects for the origin and paraphrasing tasks, potentially stemming from unknown overlaps in its pretraining data. However, these benefits do not extend to the mutation and code-rewriting tasks, for which Qwen2.5-Coder-7B provides only limited accuracy increases. This discrepancy indicates how instruction tuning can lead to task-specific memorization, but fail to generalize across different types of code transformation.

We also observe that Qwen2.5-Coder-7B performs roughly 20% better on the origin dataset than on the code-rewriting and mutation datasets in the Table 1. Interestingly, the AST similarity shows that, on the code-rewriting dataset, the model's responses more resemble the canonical solution from rewriting set itself than those from the original set. The notable accuracy drop and largest red memorization score in Table 1 indicates that the generated code—despite its plausible code logic and structure—fails to replicate the intended functionality. Consequently, the model appears to rely on memorized patterns from the original training data rather than truly understanding the rewriting task.

There are also some interesting findings about these baseline models in Table 1. Mutation evolution tasks are particularly challenging for LLMs, likely because this transformation is rarely encountered in code corpora. Such minor textual perturbations could confuse the model, resulting in performance drop. By contrast, accuracy on the para-

Epoch	$Acc(\mathcal{T})$	$Acc(\mathcal{T}_{\text{rew}})$	$Acc(\mathcal{T}_{\text{mut}})$	$Acc(\mathcal{T}_{\text{par}})$	$Acc(\mathcal{T}_{\text{valid}})$	$sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$	$sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$	Mem($\mathcal{T}$)
baseline	0.407	0.420	0.327	0.425	0.351	0.133	0.169	0.016
20	0.393	0.190	0.323	0.376	0.140	0.364	0.176	0.286
40	0.624	0.265	0.509	0.602	0.070	0.639	0.244	0.560
60	0.695	0.274	0.602	0.677	0.088	0.649	0.238	0.588
80	0.717	0.288	0.628	0.708	0.070	0.672	0.258	0.589
100	0.721	0.292	0.633	0.712	0.105	0.670	0.263	0.583
120	0.726	0.301	0.628	0.717	0.105	0.683	0.266	0.594

Table 3: Evaluation of Llama3.1-8B-Instruct during supervised fine-tuning. Each row shows the model's performance at different epochs. We report accuracy on the original train set ($Acc(\mathcal{T})$), and on its code-rewritting ($Acc(\mathcal{T}_{\text{rew}})$), mutation ($Acc(\mathcal{T}_{\text{mut}})$), and paraphrasing ($Acc(\mathcal{T}_{\text{par}})$) variants, as well as on the validation set ($Acc(\mathcal{T}_{\text{valid}})$). We also include similarity scores between the original and code-rewritting train sets ($sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$), within the code-rewritting train set ($sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$), and a memorization score for the original train set (Mem($\mathcal{T}$)). We highlighted (red) the epoch at which the validation loss starts to rise substantially (onset of overfitting)

Methods	$Acc(\mathcal{T})$	$Acc(\mathcal{T}_{\text{rew}})$	$Acc(\mathcal{T}_{\text{mut}})$	$Acc(\mathcal{T}_{\text{par}})$	$sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$	$sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$	Mem($\mathcal{T}$)
BASELINE	0.615	0.442	0.473	0.628	0.157	0.230	0.100
SFT_W_REWRITING	0.686	0.438	0.624	0.668	0.397	0.301	0.204
SFT_W_HALF&HALF	0.544	0.403	0.438	0.527	0.499	0.234	0.354
DPO	0.465	0.482	0.354	0.491	0.101	0.159	0.022
PROBLEM TRANSLATION	0.447	0.438	0.389	0.465	0.180	0.277	0.016

Table 4: Evaluation of mitigation methods on Qwen2.5-Coder-7B. Each row shows the model's performance at different metigation methods. We denote finetuning with code-rewriting dataset as SFT_W_REWRITING, and finetuning with half-half origin-rewriting dataset as SFT_W_HALF&HALF. We report accuracy on the original train set ($Acc(\mathcal{T})$), and on its code-rewritting ($Acc(\mathcal{T}_{\text{rew}})$), mutation ($Acc(\mathcal{T}_{\text{mut}})$), and paraphrasing ($Acc(\mathcal{T}_{\text{par}})$) variants, as well as on the validation set ($Acc(\mathcal{T}_{\text{valid}})$). In addition, we include similarity scores between the original and code-rewritting train sets ($sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$), within the code-rewritting train set ($sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$), and a memorization score for the original train set (Mem($\mathcal{T}$)). For the two sft methods, we also choose the results of early-stage memorization before overfitting.

phrasing dataset remains stable or even increases for both Qwen2.5 and Llama3.1. This result indicates that rephrasings do not hamper—and may even facilitate—comprehension.

Taken together, these findings shows a divergence in how LLMs handle different types of data perturbation: purely semantic transformations can be more accessible, whereas structural or syntactic modifications may expose the models' reliance on memorization rather than deeper code understanding.

4.2.2 Memorization Becomes Stronger during Fine-tuning Process

In Figure 3, we plot the training and evaluation loss curves, from which we *define* our method to distinguish early-stage and late-stage memorization. Specifically, we designate the point at which the validation loss starts to rise substantially (epoch 20) as the onset of overfitting, and then we track subsequent changes in accuracy, AST similarity, and our proposed memorization score to assess how severely the model relies on training examples.

Tables 2 and 3 present the supervised fine-tuning results of Qwen2.5-Coder-7B and Llama-3.1-8B-Instruct, respectively. Focusing on the epoch before which overfitting occurs in both tables (highlighted on red), we observe a wide accuracy gap between the original training set and the code-rewriting set. Meanwhile, the AST-similarity gap transitions from negative to positive, implying that the model outputs for the rewritten inputs begin to resemble the original training solutions more closely than those of the rewritten dataset. This indicates a dependence on memorized training patterns, which could be further supported by a significant increase in the memorization score.

In contrast, the accuracies on the mutation and paraphrasing datasets (Tables 2 and 3) continue to rise. Because the canonical solution of mutation and paraphrasing dataset stays the same as train dataset, fine-tuning can help model memorize more on the train dataset, finally improve other two accuracies. However, an enduring accuracy gap between the training set and the mutation set reflects the model's limited problem-solving capability and its reliance on memorization rather than

solid comprehension.

Furthermore, we propose using the memorization score as a novel metric for detecting overfitting without the need for a validation dataset. By examining the performance after overfitting (epoch 20 in Table 2 and epoch 40 in Table 3), we observe that the score of late-stage memorization increases much slower than it does before overfitting. Two factors could explain this slowdown: (1) the accuracy gap between the training and mutation/paraphrasing datasets narrows, restricting further expansion in the accuracy-based portion of the memorization score; and (2) once overfitting sets in, the model cannot recover strong performance on the validation set, thus capping additional gains in the AST-similarity gap. Consequently, the memorization score's increment remains significantly lower than it was before overfitting.

4.2.3 RL and Problem Translation Help to Mitigate Memorization but Compromise the Performance

We first choose DPO among the RL-based method in Table 4. After the DPO process, we find the memorization score drops to a quite small number (0.022). While this indicates a mitigation of the memorization phenomenon, it's important to note that this improvement comes at the cost of the model's overall capabilities. Although the accuracy on the code-rewriting dataset remains stable, performance on the other three datasets declines significantly, particularly for the original training dataset, which sees a drop of about 20%. We set the origin dataset as the loser in DPO process, leading DPO to prefer the code-rewriting dataset to the origin one. This accounts for the performance drop on the original dataset. However, the performance improvement on the code-rewriting dataset is modest, suggesting that we should not treat the original and evolved datasets as entirely separate entities, as DPO does.

For the problem translation method, we observe similar results as DPO: lower memorization score and lower accuracy on origin dataset in Table 4. We suppose two situations: First, sometimes there is no difference between origin and translated problem, but the target model still tries to distinguish difference. Second, the professional model is not always correct in fact, so the translated problem and difference may be wrong. Therefore, the additional difference may make the origin problem more complex, and even mislead the target LLM. Consequently, both DPO and problem translation have their drawbacks, and they are not perfect methods to mitigate memorization in code generation.

Besides, we can find supervised fine-tuning is not be able to decrease the memorization score. As is shown in Table 4, we choose two combinations of dataset to fine-tune on: the full code-rewriting dataset (SFT_W_REWRITING) and the half-half origin-rewriting dataset (SFT_W_HALF&HALF). No matter which dataset we choose, the accuracy on the code-rewriting dataset all drops while the accuracy on the origin train dataset rises. This will obviously increase the accuracy gap. At the same time, the AST similarity gap becomes positive and even higher after fine-tuning, which means the generated code are more similar to the canonical solution of origin dataset. Moreover, the memorization score of half-half origin-rewriting dataset is higher than that of total code-rewriting dataset, which suggests that the more exposure of origin train data in fine-tuning, the more serious memorization will be.

5 Contributions

In this work, we investigated memorization in LLM code generation, where models generate correct solutions for training tasks but struggle with variant tasks. We introduced a multi-level evolution framework, transforming programming problems through mutation, paraphrasing, and code rewriting to test if models grasp problem-solving logic instead of recalling training examples. We proposed a memorization score based on accuracy differences and AST similarity gaps to gauge code memorization. Our experiments show that code-specialized LLMs (e.g., Qwen2.5-Coder-7B) tend to memorize more, scoring well on original datasets but dropping in performance on rewritten tasks. We explored mitigation strategies—supervised fine-tuning, reinforcement learning, and problem translation—finding they reduce memorization at the cost of lower performance on the original dataset. We hope our research sheds light on the reasons for performance drops on evolved datasets.

6 Limitation

While our multi-level evolution framework and memorization score offer an effective evaluation of memorization in LLM code generation, several limitations require further exploration:

(1) Memorization Solution: We try three common mitigation methods, but they either compro-

mise the performance or fail to alleviate the memorization issue. And it suggests that a better solution is needed to mitigate memorization without compromising the performance.

(2) Score Threshold for Overfitting: The rate of increase in our memorization score can approximately indicate the onset of overfitting, but it lacks accuracy and reliability. Establishing a specific score threshold would be more effective in detecting overfitting without the need for a validation dataset.

(3) Code Complexity: The code solutions in the MBPP-Plus dataset are relatively short and simple, making them easier for LLMs to memorize. Using a dataset with more complex code could yield different results.

These limitations highlight the importance of ongoing research and development efforts aimed at addressing the challenges associated with memorization in LLM code generation.

7 Ethnic Statement

The development of our multi-level evolution and memorization are guided by ethical principles to ensure responsible and beneficial outcomes.

(1) Data: Our dataset is constructed from MBPP-Plus dataset, which guarantees ethnic fairness. We actively work to eliminate any harmful or offensive content from the evolved datasets to mitigate potential risks.

(2) Responsible Usage and License: The use of these prompts and codes is intended solely for evaluating memorization in large language model (LLM) code generation tasks, with the aim of advancing scientific knowledge in the field. We encourage the responsible use of the evolved dataset for educational, scientific, and creative purposes, while strongly discouraging any harmful or malicious activities.

(3) AI Usage: Apart from the evolution process, during paper writing, we only use AI agents like GPT-4o to correct semantic errors in specific sentences.

References

AnubisLMS. 2023. Mayat. https://github.com/AnubisLMS/Mayat. Accessed: 2025-02-04.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Reza Bayat, Mohammad Pezeshki, Elvis Dohmatob, David Lopez-Paz, and Pascal Vincent. 2024. The pitfalls of memorization: When memorization hurts generalization. *Preprint*, arXiv:2412.07684.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2023. Quantifying memorization across neural language models. *Preprint*, arXiv:2202.07646.

Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The secret sharer: Evaluating and testing unintended memorization in neural networks. *Preprint*, arXiv:1802.08232.

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. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Aryan Gulati, Brando Miranda, Eric Chen, Emily Xia, Kai Fronsdal, Bruno de Moraes Dumont, and Sanmi Koyejo. 2024. 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*.

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.

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. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large

language models for code generation. *Preprint*, arXiv:2406.00515.

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. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Zhi-Cun Lyu, Xin-Ye Li, Zheng Xie, and Ming Li. 2024. Top pass: Improve code generation by pass@k-maximized code ranking. *Preprint*, arXiv:2408.05715.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Avi Schwarzschild, Zhili Feng, Pratyush Maini, Zachary C. Lipton, and J. Zico Kolter. 2024. Rethinking llm memorization through the lens of adversarial compression. *Preprint*, arXiv:2404.15146.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. *Preprint*, arXiv:2212.10560.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. *Preprint*, arXiv:2312.02120.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *Preprint*, arXiv:2403.19114.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. 2024. On memorization of large language models in logical reasoning. *Preprint*, arXiv:2410.23123.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. 2024. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM.

Chiyuan Zhang, Daphne Ippolito, Katherine Lee, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. 2023. Counterfactual memorization in neural language models. *Preprint*, arXiv:2112.12938.

Appendix

A Case Study

The result of case study comes from Qwen2.5-Coder-7B. We first focus on this problem, which is calculating the area of a rectangle:

- origin problem: Write a function to find the area of a rectangle.
- mutation problem: wrITE a fUnCTIon T0 fnid teh area oF A R3cT4nglE.
- paraphrasing problem: Create a function that calculates the area of a rectangle.
- code-rewriting problem: Write a function to find the area of a rectangle, where the calculation uses the perimeter and one side to determine the area.

As is shown in Figure 4, the original code solution takes the lengths of two sides as input, while the rewriting code solution uses the perimeter and one side as input.

A.1 Baseline and SFT Models Encounter Severe Memorization

For the origin, mutation, and paraphrasing datasets, we observe that all models can answer correctly, with responses are even the same as the solution to the original problem. We attribute this to the simple structure of the original problem's solution, which makes it easy to memorize and thus achieve high performance as is shown in Table 1, 2.

In contrast, for the code-rewriting problem, as shown in Figure 5, the baseline and supervised fine-tuning models mistakenly assume that the difference between the perimeter and twice the side length equals the length of the other side. In fact, the correct length should be half of this difference. Consequently, while the model can solve the original problem correctly, it fails to provide the correct answer for the code-rewriting problem, illustrating a memorization phenomenon in the context of the original problem.

A.2 RL and Problem Translation Help Mitigate Memorization

We first notice that supervised fine-tuning with both code-rewriting dataset and half-half origin-rewriting dataset still provide a wrong code response in Figure 5. They even multiply the perimeter and side together, which is meaningless in mathematics. Thus we argue the supervised fine-tuning methods can't help mitigate the memorization phenomenon, which is also proved by their higher memorization score in Table 4.

When it comes to the translation and DPO, their responses are actually correct. Therefore, we suggest both translation and DPO can help mitigate this memorization phenomenon. This result is highly aligned with their lower memorization score in Table 4.

B Dataset

The MBPP-Plus dataset (Liu et al., 2023) is based on the MBPP dataset (Austin et al., 2021), with 378 selected programming tasks. Each task has approximately 35 times more test cases than the original MBPP, so it offers a more rigorous evaluation of code robustness. And when it comes to the generation part, we set the random seed as 0, the max_token as 1024 to enable greedy sampling with enough context length.

B.1 Code-Rewriting Dataset

For each task in the MBPP-Plus dataset, we generated a corresponding *rew* version. We used GPT-4o to perform the rewrite process. First, as illustrate in Figure 1, we guide GPT-4o to rewrite the logic and structure of the original code z such as altering conditional statements or changing loop iterations. The rewritten code must differ semantically from the original to avoid superficial changes like variable renaming. Then, we input the original prompt x_{rew} and solution into GPT-4o, then the model generates a new prompt based on the rules defined above. An example of this rewritten process is illustrated in Figure 1.

B.2 Mutation Dataset

We created a *mutation* version by introducing controlled textual noise. Unlike the rewriting approach, which alters the code logic and structure, the mutation strategy focuses on surface-level transformations that preserve the original meaning while changing the textual appearance. Specifically, For task description x, as illustrated in Figure 1, we instruct GPT-4o to reorder its characters or fragments within words, capitalize letters at unpredictable positions, and inject or substituting characters (e.g., adding punctuation marks or swapping letters) to simulate noisy text input. An example of this mutation process is illustrated in Figure 1.

Epoch	$Acc(\mathcal{T})$	$Acc(\mathcal{T}_{\text{rew}})$	$Acc(\mathcal{T}_{\text{mut}})$	$Acc(\mathcal{T}_{\text{par}})$	$Acc(\mathcal{T}_{\text{valid}})$	$sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$	$sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$	$\text{Mem}(\mathcal{T})$
baseline	0.615	0.442	0.473	0.628	0.561	0.157	0.230	0.100
10	0.654	0.367	0.566	0.65	0.561	0.295	0.217	0.200
20	0.823	0.429	0.739	0.814	0.474	0.499	0.234	0.428
30	0.863	0.398	0.805	0.845	0.544	0.571	0.277	0.474
40	0.881	0.416	0.801	0.867	0.509	0.56	0.277	0.469
50	0.938	0.429	0.836	0.907	0.474	0.605	0.275	0.544
60	0.942	0.429	0.841	0.925	0.456	0.612	0.279	0.544
70	0.951	0.447	0.858	0.947	0.509	0.624	0.287	0.538
80	0.947	0.442	0.858	0.947	0.509	0.623	0.285	0.540
90	0.947	0.447	0.858	0.947	0.509	0.624	0.286	0.534
100	0.947	0.447	0.863	0.951	0.526	0.623	0.284	0.532
110	0.951	0.447	0.863	0.951	0.509	0.62	0.285	0.533
120	0.947	0.447	0.863	0.951	0.526	0.62	0.283	0.530

Table 5: Full results of evaluation of Qwen2.5-Coder-7B during supervised fine-tuning. Each row shows the model's performance at different fine-tuning epochs. We report accuracy on the original train set ($Acc(\mathcal{T})$), and on its code-rewritting ($Acc(\mathcal{T}_{\text{rew}})$), mutation ($Acc(\mathcal{T}_{\text{mut}})$), and paraphrasing ($Acc(\mathcal{T}_{\text{par}})$) variants, as well as on the validation set ($Acc(\mathcal{T}_{\text{valid}})$). In addition, we include similarity scores between the original and code-rewritting train sets ($sim_{\mathcal{T}}(\mathcal{T}_{\text{rew}})$), within the code-rewritting train set ($sim_{\mathcal{T}_{\text{rew}}}(\mathcal{T}_{\text{rew}})$), and a memorization score for the original train set ($\text{Mem}(\mathcal{T})$). We highlighted (red) the epoch at which the validation loss starts to rise substantially (onset of overfitting)

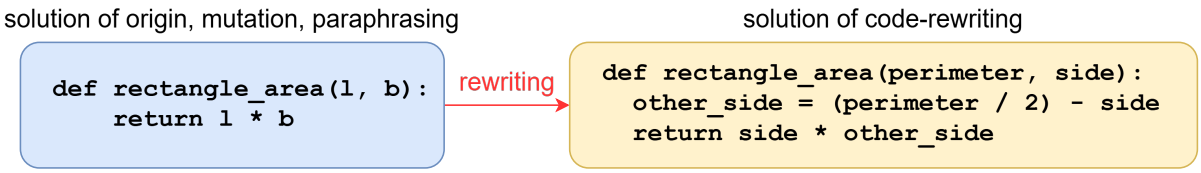

Figure 4: The canonical solution on one origin dataset and three evolution datasets. The left blue box is the solution of origin, mutation and paraphrasing dataset, while the right yellow box is the solution of code-rewriting dataset. And the red arrow stands for the code-rewriting evolution.

B.3 Paraphrasing Dataset

In addition to code rewriting and mutation, we further expanded our dataset with a *paraphrase* version of each prompt. As illustrated in Figure 1, we prompt GPT-4o to rephrase the original input using new wording or sentence structures. The aim is to preserve the original meaning while altering its expression, effectively introducing a small noise ϵ_2 within the semantic space of natural language S. An example of this paraphrase process is illustrated in Figure 1.

C Model

We choose two series of LLM to conduct our research: Qwen2.5-7B and Qwen2.5-Coder-7B, along with Llama-3.1-8B and Llama-3.1-8B-Instruct. In order to conduct our experiments on 8*NVIDIA A100 GPUs, we choose the model size of 7B and 8B.

During the evolution process, we choose GPT-4o to conduct our evolution methods due to its strong capability and high performance on code-related tasks. In order to preserve its creativity, we set the temperature as 1 to better evolve our origin dataset. However, when it comes to response generation (inference), we set the *temperature* as 0 to ensure greedy sampling for certainty.

baseline / sft_w_origin

```
def rectangle_area(perimeter, side):
    return (perimeter - 2*side) * side
```

sft_w_rewriting / sft_w_half&half

```
def rectangle_area(perimeter, side):
    return (perimeter * side) / 2
```

problem translation

```
def rectangle_area(perimeter, side):
    return (perimeter - 2 * side) / 2 * side
```

DPO

```
def rectangle_area(perimeter, side):
    other_side = (perimeter - 2 * side) / 2
    area = side * other_side
    return area
```

Figure 5: The response of code-rewriting problem. The red box stands for the wrong response, and the green box stands for the correct response. The top-left box is the response of baseline model and supervised fine-tuned model with origin dataset, while the top-right box is the response of supervised fine-tuned model with code-rewriting and half&half dataset in the mitigation process. The bottom box is the response of translation and DPO models.