Beyond Benchmark Scores: Evaluating Robustness and Memorization in LLM Code Generation with Evolved Questions

Anonymous ACL submission

Abstract

With the rapid advancement in large language models (LLMs), their ability in code generation has received significant attention. Evaluating this capability involves assessing both **robustness** and **memorization** behaviors. Robustness expects that syntactic modifications of a prompt—without changing its semantics—should produce functionally equivalent generated code. Conversely, memorization occurs when an LLM produces code very similar to solutions seen during training, even when the prompt's meaning has semantically changed. In this paper, we systematically investigate these phenomena by introducing three prompt-variation strategies: **mutation** (minor textual noise), **paraphrasing** (different wording, same meaning), and **code-rewriting** (similar wording, different meaning). Based on these strategies, we propose two metrics to quantify these behaviors: Robustness Ratio (RR), measuring how consistently models solve tasks despite textual perturbations, and Memorization Risk Index (MRI), capturing how often models reproduce known solutions despite semantic prompt changes. Our experiment illustrates that as task complexity increases and model size decreases, robustness generally declines. Additionally, supervised fine-tuning (SFT) significantly improves origin accuracy but often at the expense of increased memorization, while proximal policy optimization (PPO) provides a more balanced trade-off.

1 Introduction

Large language models (LLMs) have made incredible advances in automated code generation, and are rapidly becoming essential tools in software development. Modern code-focused LLMs can achieve state-of-the-art performance on programming benchmarks (Rozière et al., 2024). For example, specialized models like Qwen-2.5 Coder(Hui et al., 2024) and Code Llama (Rozière et al., 2024) have pushed the boundaries of translating natural language into code (Hui et al., 2024). However, questions remain about how well these models generalize beyond their training data, specifically in terms of robustness and memorization.

Robustness in LLMs refers to a model's capacity to consistently deliver accurate and reliable outputs across varied scenarios, ensuring performance stability even when queries are phrased differently or posed within diverse contexts. In the context of code generation, as suggested by (Sarker et al., 2024) robustness evaluates the degree to which the semantically equivalent prompts elicit semantically equivalent responses.

Memorization occurs when LLMs rely on recalling specific training examples rather than genuine generation (Hartmann et al., 2023), (Lu et al., 2024). In the code generation domain, as suggested by (Yang et al., 2024), memorization evaluates the extent to which an LLM reproduces code solutions from its training data when systematically exposed to prompts that differ semantically from the original but still similar in text.

In this paper, we evaluate robustness and memorization as two key dimensions of generalization. We introduce a multilevel evolution framework that generates variant coding tasks using three distinct evolution methods: *mutation*, *paraphrasing*, and *code-rewriting*. Specifically, the *code-rewriting* method creates semantically modified but superficially similar tasks, while *paraphrasing* and *mutation* generate tasks that preserve semantics but differ significantly at the textual level. We further introduce two key metrics: Robustness Ratio (RR), quantifying how robust models are to prompt variations, and Memorization Risk Index (MRI), assessing how likely models reproduce memorized solutions after semantic modifications to tasks.

Our evaluation framework examines whether LLMs can sustain their performance on these variant tasks. To ensure comprehensive insights across varying difficulty levels, our assessment

covers from introductory programming problems in MBPP+ (Liu et al., 2023) to complex, multi-library scenarios BIGCODEBENCH (Zhuo et al., 2024). Furthermore, we finetune several base models—including Qwen-2.5-7B and Qwen-2.5-Coder 7B—using two popular fine-tuning strategies: Supervised Fine-Tuning (SFT) and Proximal Policy Optimization (PPO) to investigate the effect of post-training technique on robustness and memorization of models.

Contributions. In summary, our work makes the following key contributions:

- We propose a novel multi-level evolution framework that systematically generates diverse task variants (mutation, paraphrasing, code-rewriting), along with two metrics: Robustness Ratio (RR) for robustness evaluation and Memorization Risk Index (MRI) for detecting memorization. These tools provide a better understanding of LLM robustness and memorization behaviors.
- We conduct extensive evaluation of robustness and memorization spanning two established benchmarks: MBPP+, consisting of simple introductory Python problems, and BIGCODEBENCH, comprising complex, multi-library coding scenarios that reflect real-world software tasks.
- Our comprehensive analysis of SFT and PPO offers key insights into fine-tuning trade-offs: SFT significantly improves raw accuracy but increases memorization risk, while PPO achieves a better balance between accuracy and memorization.

2 Related Work

2.1 Code Generation with LLMs

Large Language Models (LLMs) have shown remarkable ability in automated code generation. Models such as ChatGPT(OpenAI et al., 2024), Qwen-Coder (Hui et al., 2024), and DeepSeek-Coder (Guo et al., 2024) have pushed the boundaries in the coding domain, notably with ChatGPT achieving state-of-the-art performance on challenging benchmarks such as BigCodeBench (Zhuo et al., 2024) and the EvalPlus leaderboard (Liu et al., 2023).

While LLM-based code generation models have made significant strides in translating natural language to executable code, most evaluations focus on static benchmark performance, overlooking robustness to prompt variations and memorization behaviors. To address this, we introduce a novel multi-level evolution framework, enabling systematic assessment of these critical generalization aspects.

2.2 Robustness and Memorization in Code Generation.

Robustness Robustness refers to the phenomenon that when minor changes are applied to input, the LLM fails to generated consistent and correct responses (Li et al., 2022). Besides, metamorphic prompt testing (Wang and Zhu, 2024) has proved that LLM code outputs are highly sensitive to how problems are presented.

Memorization Memorization refers to the LLM will provide the exact text snippet in the training data (Carlini et al., 2019), (Bayat et al., 2024). In the domain of code generation, new methods (Wang et al., 2024) have been proposed to measure and expose code memorization. Moreover, Liu et al. (2023) shows many LLM outputs that passed the original tests fail the larger suite. Together, these analyses reveal that direct copying of training code is common, and that standard evaluations and simple prompts often fail to detect it.

3 Methodology

3.1 Multi-level Evolution

We propose a multi-level evolution framework to systematically assess robustness and memorization in LLM-generated code. Figure 1 illustrates our multi-level evolution methods. We categorize task evolution into two scenarios based on whether the semantic remains consistent. Specifically, *mutation* and *paraphrasing* evolutions **preserve semantic equivalence**, whereas *code-rewriting* evolution generates **semantically different** tasks.

Mutation Evolution. To assess whether LLMs are robust to superficial textual noise, mutation evolution applies small perturbations—such as word-scrambling, random-capitalization, and character-noising—that preserve the underlying problem semantics. This approach simulates common input variations encountered in real-world scenarios, testing the model's ability to generate correct code even when prompts are noisy or non-standard. Formally, let $x \in T$ denote the original problem prompt in the text space T. Mutation evolution applies a perturbation function $\epsilon_1 : T \to T$ such that the mutated prompt $x_{mut} = \epsilon_1(x)$ preserves

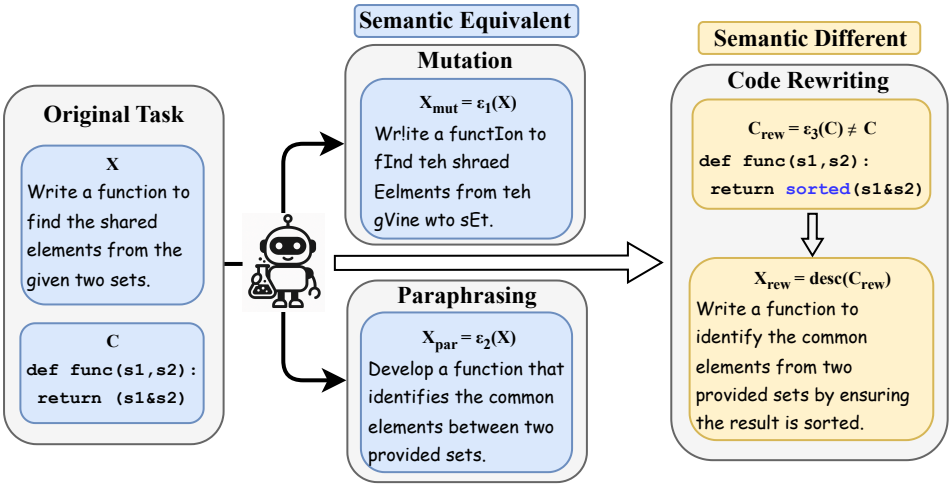

Figure 1: **The multi-level evolution framework workflow**. "X" denotes text and "C" denotes code. Same-colored boxes indicate semantic equivalence. Mutation and Paraphrasing evolution keep the same semantic, while code rewriting creates a new task. Code rewriting first rewrite a new code solution C_{rew} from the origin solution C, then generating a new description X_{rew} for the new code. All evolutions are performed by GPT-4.1, shown as the robot.

the original semantics:

$$x_{mut} = \epsilon_1(x), \quad x, x_{mut} \in T$$

where ϵ_1 injects textual noise without altering the problem's underlying meaning.

Paraphrasing Evolution. Paraphrasing evolution aims to evaluate whether LLMs can generalize to diverse surface realizations of the same problem. In this setting, prompts are reworded textual expression but preserve semantics. Formally, let $x \in T$ be the original prompt. We define a paraphrasing function $\epsilon_2 : T \to T$ such that:

$$x_{par} = \epsilon_2(x), \quad x, x_{par} \in T$$

where x_{par} is a semantically equivalent but textually different paraphrase of x. This process tests whether LLMs can solve the same programming problem across varied linguistic formulations.

Code-Rewriting Evolution. Code-rewriting evolution evaluates a model's ability to distinguish new problem semantics that are superficially similar to origin tasks. Specifically, we first modify the ground truth solution while preserving the original function signature—including the function name, input, and output format. We then generate a new task description that reflects the altered code. Formally, let $x \in T$ be the original prompt and $c \in C$ its ground truth code solution in code space C. We apply a rewriting function ϵ_3 that produces a new code $c_{rew} = \epsilon_3(c)$ where $c_{rew} \neq c$ functionally but both c and c_{rew} share the same signature. The new prompt x_{rew} is then generated from c_{rew}, resulting in a semantically different task:

$$x_{rew} = \text{desc}(c_{rew})$$
$$\text{where} \quad \text{sig}(c_{rew}) = \text{sig}(c), \quad c_{rew} \neq c$$

where $\text{desc}(\cdot)$ denotes generating a description from code, and $\text{sig}(\cdot)$ extracts the function signature. This process enables us to assess whether LLMs can recognize and solve tasks that share format but differ in semantic content.

Task Generation Details. All evolution tasks—mutation, paraphrasing, and code-rewriting—are automatically generated using GPT-4.1. The full prompts for generating these task variants are included in Appendix A.

3.2 Evaluation Metrics

3.2.1 Robustness

(1) Pass@1 Accuracy For a task set $\mathcal{T}$, Pass@1 is the fraction of tasks for which the model's top-1 prediction passes all unit tests. We report it as $\text{Acc}(\mathcal{T})$. It is the standard measure in code-generation work.

(2) Robustness Ratios. For each transformation type $\phi \in [mut, pra]$ (mut for mutation and par for paraphrase), we compute a **Robustness Ratio (RR)**:

$$\text{RR}_\phi = \min\Big(1, \frac{\text{Acc}(\mathcal{T}_\phi)}{\text{Acc}(\mathcal{T}_{\text{ori}})}\Big) \tag{1}$$

which asks: once we perturb a prompt without changing its semantics, what fraction of previously-solved tasks remain solved?
The cap at 1 makes the score comparable across models: values below 1 indicates performance drop, while values above 1 are truncated to avoid inflating means. This ratio formulation has two advantages: (i) it is unit-free, letting us compare models with different base accuracies, and (ii) it decouples absolute skill (Acc) from stability under perturbation.

Reporting. For every model we present

$$\big[\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}}),\ \mathrm{RR}_{\mathrm{mut}},\ \mathrm{RR}_{\mathrm{par}}\big]$$

to distinguish robustness from *(i)* character-level mutations and *(ii)* paraphrastic rewrites.

3.2.2 Memorization

A model *memorizes* when it reproduces code that solves the **original** task but fails once the underlying semantics are changed by *code-rewriting*(§3.1). The metric thus integrates two signals:

- Similarity between the model-generated solution for the rewritten task and the original ground-truth solution, and
- Performance drop by semantic rewriting.

(1) Solution–Similarity. For every rewritten task $i \in \mathcal{T}_{\mathrm{rew}}$, we measure

- AST similarity: normalised tree-edit overlap between abstract-syntax trees (1 = identical structure, 0 = unrelated)
- Edit similarity: 1 - (Levenshtein distance / max-len), capturing token-level overlap.

Formally, let $\mathrm{AST}_i \in [0,1]$ denote AST similarity, and let $\mathrm{Edit}_i \in [0,1]$ denote edit similarity. We combine these scores into a unified similarity score:

$$S_i = \frac{\mathrm{AST}_i + \mathrm{Edit}_i}{2}$$

Because our analysis is corpus-level, we define the mean similarity over all rewritten tasks as:

$$\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}}) = \frac{1}{|\mathcal{T}_{\mathrm{rew}}|} \sum_{i \in \mathcal{T}_{\mathrm{rew}}} S_i, \quad \mathrm{Sim} \in [0,1].$$

(2) Relative Accuracy Drop (RAD). To capture the performance loss induced by semantic rewriting, we define RAD as the relative reduction in accuracy from original tasks to rewritten tasks:

$$\mathrm{RAD} = \max\left(0,\ \frac{\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}}) - \mathrm{Acc}(\mathcal{T}_{\mathrm{rew}})}{\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})}\right)$$

where $\mathrm{Acc}(\cdot)$ is computed as defined in Eq. (3.2.1), and $\mathrm{RAD} \in [0,1]$. RAD is 0 when meaning changes do not hurt accuracy and positive when they do. The $max(0,)$ prevents negative values when a rewritten task happens to be easier.

(3) Memorization Risk Index (MRI). Finally, we introduce the MRI, defined as the product of solution-similarity and relative accuracy drop:

$$\mathrm{MRI} = \mathrm{Sim}(\mathcal{T}_{\mathrm{rew}}) \ \times\ \mathrm{RAD}, \quad \mathrm{MRI} \in [0,1].$$

Product makes MRI is high only when both conditions for harmful memorization hold: (i) the model copies the original solution's surface form (high Sim) and (ii) that copied solution now fails (high RAD). This multiplicative design sharply distinguishes superficial recall from generalization.

Reporting. For every model, we report:

$$\big[\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}}),\ \mathrm{Acc}(\mathcal{T}_{\mathrm{rew}}),\ \mathrm{RAD},\ \mathrm{Sim}(\mathcal{T}_{\mathrm{rew}}),\ \mathrm{MRI}\big].$$

to assess model coding proficiency, sensitivity to semantic changes, similarity-based copying tendencies, and overall memorization risk separately and comprehensively.

3.3 Fine-Tuning Methods

To investigate the memorization phenomenon, we use the original data for finetuning. Additionally, to assess the impact of fine-tuning, we select base models as the foundation for this process.

3.3.1 Supervised Finetuning

Supervised Finetuning 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 serve as the inputs, while code solutions act as the corresponding labels. However, overfitting occurs when the model fits the training data too closely, reducing its ability to generalize to unseen tasks. This is typically indicated by a rise in validation loss, signaling that the model has begun memorizing training examples rather than capturing the underlying task principles. Therefore, we distinguish between early-stage and

late-stage memorization. Late-stage memorization aligns closely with traditional overfitting, so we select the checkpoint just before overfitting sets in, ensuring a clearer distinction between early-stage memorization and overfitting.

3.3.2 Reinforcement Learning

In the large language model era, Reinforcement Learning enhances fine-tuning efficiency. A leading method is Proximal Policy Optimization (PPO)(Schulman et al., 2017), which alternates between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. We utilize the same model architecture for the actor, critic, and reference models, and define the reward function based on the correctness of the generated code. Compared to other reinforcement learning methods like DPO (Rafailov et al., 2024), we suggest that using accuracy as the reward function offers a more direct and efficient optimization path.

4 Experiments

4.1 Datasets

We conduct our evaluation on two widely-adopted code generation benchmarks: **MBPP+** (Liu et al., 2023) and **BigCodeBench** (Zhuo et al., 2024).

Dataset Statistics. MBPP+ contains 378 tasks, and BigCodeBench comprises 1140 tasks. We use 4:1 train/test split for fine-tuning. For models without fine-tuning, we use the **complete set** of benchmark tasks for evaluation. For models that undergo SFT and PPO, we train on the **training split** and evaluate on the **test split**.

Task Evolution. For each task in the test set, we generate three types of variants using our multi-level evolution framework 3.1. For each original task, we generate one variant for each evolution type, resulting in four versions per task: original, mutation, paraphrased, and code-rewritten. More about generating evolved datasets can be found in the appendix D.

Data Validation. To ensure the reliability of our evolved datasets, we conducted a manual quality assurance process. Two senior researchers with expertise in the field randomly reviewed 10% generated evolution problems for each dataset for correctness and clarity. Any issues identified during this review were rectified by re-running the generation scripts and re-assessing the outputs.

Data Release and Reproducibility. All evolved task variants and the prompts used for task generation will be released publicly upon publication, ensuring reproducibility and facilitating future research (see Appendix A for prompt details).

4.2 Models

In this paper, we choose Qwen-2.5 series, Qwen-2.5-Coder series, LLaMA-3.1 series and Llama-4-Maverick-17B-16E series to conduct the scale-up experiments. For fine-tuning section, we choose Qwen-2.5-7B, and Qwen-2.5-Coder-7B due to the trade-off between their performance and our computational resources. All training and inference were conducted on a server equipped with 4 NVIDIA H100 GPUs (80GB each), with a total computational budget of approximately 40 GPU hours, using PyTorch and HuggingFace Transformers.

4.3 Result Analysis

Tables 1 and 2 summarize our robustness and memorization evaluations across both datasets.

4.3.1 Robustness Analysis

1. Mutation remains more challenging, with robustness plateauing at larger model scales. Across both benchmarks, the mutation robustness ratio (RR_{mut}) is consistently lower than the paraphrase ratio (RR_{par}), confirming that character-level perturbations degrade model performance more significantly than semantic paraphrasing. For QWEN-INSTRUCT evaluated on MBPP+, RR_{mut} sharply increases from **0.85** at `0.5B` to near-optimal **0.99** at `7B`, and decreasing marginally to **0.94–0.96** at the `14-32B` scale. A similar saturation trend is evident on BIGCODEBENCH, peaking at **0.94** for the `14B` checkpoint.

2. Paraphrasing can occasionally improve accuracy. Smaller models (0.5–3B) reach $RR_{par} \geq 1.00$, meaning paraphrased prompts solve as many—or slightly more—tasks than the originals. We manually inspected 10 random prompt pairs where accuracy improved and found that 6 contained clearer imperative wording (see examples in B. We therefore conjecture that GPT-4.1 paraphrasing could reduce prompt ambiguity, benefiting weaker models. Larger checkpoints ($\geq$7B) already parse the original phrasing well, and thus RR_{par} stabilises around 0.95–1.00. A controlled user-study or ambiguity annotation would be required to confirm this explanation.

Model	$\text{Acc}(\mathcal{T}_{\text{ori}})\uparrow$	$\text{Acc}(\mathcal{T}_{\text{mut}})\uparrow$	$\text{Acc}(\mathcal{T}_{\text{par}})\uparrow$	$\mathcal{RR}_{\text{mut}}\uparrow$	$\mathcal{RR}_{\text{par}}\uparrow$	$\text{Acc}(\mathcal{T}_{\text{rew}})\uparrow$	$\text{Sim}(\mathcal{T}_{\text{rew}})\downarrow$	RAD $\downarrow$	MRI $\downarrow$
Qwen-2.5-0.5B-Instruct	0.36	0.30	0.39	0.85	1.00	0.30	0.26	0.15	0.04
Qwen-2.5-1.5B-Instruct	0.55	0.51	0.57	0.93	1.00	0.52	0.17	0.04	0.01
Qwen-2.5-3B-Instruct	0.59	0.54	0.59	0.91	1.00	0.59	0.15	0.00	0.00
Qwen-2.5-7B-Instruct	0.65	0.64	0.64	0.99	0.99	0.62	0.14	0.04	0.01
Qwen-2.5-14B-Instruct	0.66	0.63	0.63	0.96	0.97	0.63	0.16	0.03	0.01
Qwen-2.5-32B-Instruct	0.70	0.66	0.69	0.94	0.97	0.70	0.15	0.00	0.00
Qwen-2.5-coder-0.5B-Instruct	0.39	0.34	0.44	0.87	1.00	0.33	0.21	0.14	0.03
Qwen-2.5-coder-1.5B-Instruct	0.54	0.49	0.56	0.90	1.00	0.48	0.26	0.12	0.03
Qwen-2.5-coder-3B-Instruct	0.61	0.55	0.60	0.90	0.99	0.57	0.25	0.05	0.01
Qwen-2.5-coder-7B-Instruct	0.64	0.61	0.65	0.94	1.00	0.63	0.24	0.02	0.01
Qwen-2.5-coder-14B-Instruct	0.68	0.64	0.68	0.95	1.00	0.66	0.21	0.03	0.01
Qwen-2.5-coder-32B-Instruct	0.69	0.65	0.69	0.95	1.00	0.66	0.17	0.03	0.01
Llama-3.1-8B-Instruct	0.55	0.52	0.56	0.95	1.00	0.54	0.13	0.00	0.00
Llama-3.1-70B-Instruct	0.58	0.53	0.56	0.91	0.96	0.64	0.22	0.00	0.00
Llama-4-Maverick-17B-16E	0.51	0.46	0.53	0.92	1.00	0.57	0.33	0.00	0.00
Llama-4-Scout-17B-16E-Instruct	0.62	0.58	0.61	0.94	0.97	0.61	0.14	0.00	0.00

Table 1: **Evaluation of scale-up models on MBPP+.** $\mathcal{T}_{\text{ori}}, \mathcal{T}_{\text{mut}}, \mathcal{T}_{\text{par}}, \mathcal{T}_{\text{rew}}$ denote the origin, mutation, paraphrasing and code-rewriting datasets respectively. Acc: accuracy, $\mathcal{RR}$: robustness ratio, Sim: solution similarity, RAD: relative accuracy drop, MRI: memorization risk index.

3. Task complexity influences robustness. Absolute accuracies on BIGCODEBENCH are approximately **20 percentage points** lower than those on MBPP+. Correspondingly, robustness ratios also decrease, with peak values for RRmut dropping from **0.99** to **0.94** and RRpar from **1.00** to **0.93**. Thus, tasks that are more challenging appear more vulnerable to linguistic perturbations, indicating the necessity of robustness evaluations that extend beyond simpler benchmark problems.

Additional Observations *Coder-tuned* variants marginally outperform instruction models at smaller scales (`0.5-3B`) but converge in performance at larger scales (`14-32B`). This suggests that code-specific pre-training primarily aids models yet to achieve high capacity.

In summary, model robustness improves significantly with increased parameters up to the intermediate size range (approximately `7-14B`) before reaching saturation. Paraphrasing appears less adversarial compared to mutation and may occasionally clarify task descriptions.

4.3.2 Memorization Analysis

1. Memorization declines rapidly on simpler tasks but persists on more complex scenarios. On the introductory-level tasks in MBPP+, memorization risk (MRI) decreases notably as models scale up. For instance, Qwen-2.5-Instruct's MRI falls from 0.04 at `0.5B` parameters to effectively zero at `32B`. Conversely, on the more challenging BIGCODEBENCH, MRI values remain significant even at large scales (0.09 for Qwen-2.5-32B-Instruct), indicating that memorization continues to affect complex, multi-library tasks. This discrepancy shows that while larger models better capture underlying semantics, they do not completely eliminate memorization, especially in scenarios demanding deeper reasoning.

2. Code-specific pre-training encourages code reuse but does not substantially increase memorization. Coder-specialized models consistently produce higher similarity scores than their instruction-tuned counterparts—for instance, Qwen-2.5-coder-32B-Instruct shows a similarity score of 0.35 versus 0.30 for the instruction-only variant on BIGCODEBENCH. However, the relative accuracy drop (RAD) remains comparable across these variants, translating to only a slight increase in MRI (0.11 vs. 0.09). This pattern suggests code-focused pre-training promotes superficial reuse of training snippets without significantly exacerbating harmful memorization.

3. Low RAD values reflect genuine generalization rather than memorization. Several models—including Llama variants and mid-sized Qwen models—achieve RAD scores near or exactly zero while retaining moderate solution similarities (ranging from 0.13 to 0.33). A closer manual inspection of these instances (see Appendix C) revealed that models effectively generalize by modifying only internal logic—such as substituting explicit loops with list comprehensions or switching from

Model	Acc($\mathcal{T}_{\text{ori}}$) ↑	Acc($\mathcal{T}_{\text{mut}}$) ↑	Acc($\mathcal{T}_{\text{par}}$) ↑	$\mathcal{RR}_{\text{mut}}$ ↑	$\mathcal{RR}_{\text{par}}$ ↑	Acc($\mathcal{T}_{\text{rew}}$) ↑	Sim($\mathcal{T}_{\text{rew}}$)↓	RAD ↓	MRI ↓
Qwen-2.5-0.5B-Instruct	0.09	0.05	0.10	0.58	1.00	0.04	0.27	0.55	0.15
Qwen-2.5-1.5B-Instruct	0.22	0.18	0.24	0.80	1.00	0.12	0.26	0.46	0.12
Qwen-2.5-3B-Instruct	0.31	0.25	0.30	0.81	0.97	0.18	0.31	0.42	0.13
Qwen-2.5-7B-Instruct	0.38	0.33	0.35	0.87	0.92	0.23	0.28	0.38	0.11
Qwen-2.5-14B-Instruct	0.39	0.36	0.37	0.94	0.94	0.25	0.24	0.35	0.08
Qwen-2.5-32B-Instruct	0.44	0.38	0.41	0.85	0.92	0.30	0.30	0.31	0.09
Qwen-2.5-coder-0.5B-Instruct	0.11	0.06	0.13	0.55	1.00	0.05	0.32	0.49	0.15
Qwen-2.5-coder-1.5B-Instruct	0.24	0.17	0.24	0.73	0.99	0.13	0.34	0.45	0.15
Qwen-2.5-coder-3B-Instruct	0.36	0.28	0.33	0.76	0.92	0.21	0.38	0.43	0.16
Qwen-2.5-coder-7B-Instruct	0.41	0.34	0.36	0.82	0.89	0.26	0.35	0.37	0.13
Qwen-2.5-coder-14B-Instruct	0.47	0.40	0.42	0.85	0.90	0.32	0.35	0.32	0.11
Qwen-2.5-coder-32B-Instruct	0.48	0.39	0.43	0.80	0.88	0.33	0.35	0.31	0.11
Llama-3.1-8B-Instruct	0.31	0.26	0.29	0.84	0.95	0.21	0.22	0.31	0.07
Llama-3.1-70B-Instruct	0.42	0.34	0.39	0.83	0.93	0.28	0.30	0.34	0.10
Llama-4-Maverick-17B-16E	0.42	0.36	0.38	0.86	0.90	0.25	0.28	0.38	0.10
Llama-4-Scout-17B-16E-Instruct	0.40	0.34	0.36	0.86	0.90	0.28	0.23	0.33	0.08

Table 2: **Evaluation of scale-up models on BigCodeBench.** $\mathcal{T}_{\text{ori}}, \mathcal{T}_{\text{mut}}, \mathcal{T}_{\text{par}}, \mathcal{T}_{\text{rew}}$ denote the origin, mutation, paraphrasing and code-rewriting datasets respectively. Acc: accuracy, $\mathcal{RR}$: robustness ratio, Sim: solution similarity, RAD: relative accuracy drop, MRI: memorization risk index.

slice-based lookup to binary search—while preserving all naming conventions. Such results confirm our metric appropriately distinguishes harmless syntactic similarity from problematic memorization, penalizing models only when superficial code reuse leads to functional failures.

Additional Observations Task complexity significantly impacts memorization risk. Median MRI values on BigCodeBench are roughly three times higher than those on MBPP+ (0.11 vs. 0.03), indicating the necessity of evaluating memorization across diverse and challenging tasks to get an accurate picture of a model's generalization capabilities.

4.3.3 Impact of Fine-Tuning Strategies on Robustness and Memorization

Tables 3 and 4 shows notable differences in robustness and memorization across different fine-tuning strategies.

1. SFT boosts accuracy but introduces high memorization risk. Models fine-tuned via SFT consistently achieve accuracy gains on original tasks (e.g., increasing accuracy from 0.64 to 0.66 on MBPP+ and from 0.32 to 0.39 on BigCodeBench for Qwen-2.5-7B). However, these improvements come with pronounced increases in memorization, as indicated by much higher MRI scores (0.19 on MBPP+ and 0.14 on BigCodeBench). Thus, SFT enhances surface-level accuracy at the expense of genuine generalization.

2. PPO balances accuracy improvements and memorization risk. Fine-tuning with PPO results in accuracy gains on simpler tasks (0.64 to 0.70 on MBPP+) and minor or negligible changes on more challenging datasets like BigCodeBench. Crucially, PPO maintains significantly lower memorization risks compared to SFT (MRI of 0.07 vs. 0.19 on MBPP+, and 0.05 vs. 0.14 on BigCodeBench). This suggests reinforcement learning strategies encourage models to produce correct yet novel code rather than relying on direct memorization of training examples.

3. The effect of fine-tuning on robustness varies by dataset complexity. SFT noticeably improves robustness ratios on the simpler MBPP+ benchmark (achieving robustness ratios of 1.00). Conversely, on the more complex BigCodeBench tasks, SFT provides no robustness advantage—and occasionally even reduces robustness. PPO shows relatively stable robustness ratios across both benchmarks, aligning with baseline models, implying that PPO neither significantly improves nor negatively impacts robustness.

Implications for Fine-Tuning Decisions:

- **If maximizing accuracy is the priority and the risks associated with memorization are acceptable**, then SFT remains the optimal strategy.
- **If generalization and minimizing memorization risk are critical**, PPO provides a better balance by offering modest accuracy improvements

Model	$Acc(\mathcal{T}_{\text{ori}})\uparrow$	$Acc(\mathcal{T}_{\text{mut}})\uparrow$	$Acc(\mathcal{T}_{\text{par}})\uparrow$	$\mathcal{RR}_{\text{mut}}\uparrow$	$\mathcal{RR}_{\text{par}}\uparrow$	Acc($\mathcal{T}_{\text{rew}}$) $\uparrow$	Sim($\mathcal{T}_{\text{rew}}$)$\downarrow$	RAD $\downarrow$	MRI $\downarrow$
Qwen-2.5-7B	0.64	0.59	0.64	0.92	1.00	0.59	0.20	0.09	0.02
Qwen-2.5-7B-Instruct	0.64	0.62	0.64	0.96	1.00	0.62	0.18	0.04	0.01
Qwen-2.5-7B-SFT	0.66	0.68	0.67	1.00	1.00	0.45	0.59	0.32	0.19
Qwen-2.5-7B-PPO	0.70	0.67	0.66	0.96	0.94	0.59	0.45	0.15	0.07
Qwen-2.5-Coder-7B	0.66	0.64	0.68	0.98	1.00	0.57	0.32	0.14	0.04
Qwen-2.5-Coder-7B-Instruct	0.67	0.64	0.70	0.96	1.00	0.59	0.34	0.12	0.04
Qwen-2.5-Coder-7B-SFT	0.71	0.68	0.71	0.96	1.00	0.50	0.59	0.30	0.17
Qwen-2.5-Coder-7B-PPO	0.70	0.67	0.64	0.96	0.92	0.59	0.52	0.15	0.08

Table 3: **Fine-tuning results on MBPP+.** $\mathcal{T}_{\text{ori}}, \mathcal{T}_{\text{mut}}, \mathcal{T}_{\text{par}}, \mathcal{T}_{\text{rew}}$ denote the origin, mutation, paraphrasing and code-rewriting datasets respectively. Acc: accuracy, $\mathcal{RR}$: robustness ratio, Sim: solution similarity, RAD: relative accuracy drop, MRI: memorization risk index.

Model	$Acc(\mathcal{T}_{\text{ori}})\uparrow$	$Acc(\mathcal{T}_{\text{mut}})\uparrow$	$Acc(\mathcal{T}_{\text{par}})\uparrow$	$\mathcal{RR}_{\text{mut}}\uparrow$	$\mathcal{RR}_{\text{par}}\uparrow$	Acc($\mathcal{T}_{\text{rew}}$) $\uparrow$	Sim($\mathcal{T}_{\text{rew}}$)$\downarrow$	RAD $\downarrow$	MRI $\downarrow$
Qwen-2.5-7B	0.32	0.25	0.33	0.79	1.00	0.22	0.22	0.32	0.07
Qwen-2.5-7B-Instruct	0.36	0.36	0.34	1.00	0.95	0.28	0.27	0.22	0.06
Qwen-2.5-7B-SFT	0.39	0.32	0.34	0.82	0.89	0.25	0.41	0.34	0.14
Qwen-2.5-7B-PPO	0.33	0.27	0.32	0.80	0.96	0.24	0.18	0.29	0.05
Qwen-2.5-Coder-7B	0.38	0.29	0.36	0.76	0.95	0.27	0.34	0.29	0.10
Qwen-2.5-Coder-7B-Instruct	0.42	0.31	0.36	0.74	0.86	0.28	0.35	0.34	0.12
Qwen-2.5-Coder-7B-SFT	0.42	0.35	0.40	0.84	0.96	0.26	0.42	0.38	0.16
Qwen-2.5-Coder-7B-PPO	0.35	0.29	0.34	0.83	0.98	0.27	0.42	0.24	0.10

Table 4: **Fine-tuning results on BigCodeBench.** $\mathcal{T}_{\text{ori}}, \mathcal{T}_{\text{mut}}, \mathcal{T}_{\text{par}}, \mathcal{T}_{\text{rew}}$ denote the origin, mutation, paraphrasing and code-rewriting datasets respectively. Acc: accuracy, $\mathcal{RR}$: robustness ratio, Sim: solution similarity, RAD: relative accuracy drop, MRI: memorization risk index.

while considerably reducing memorization.

5 Conclusion

In this paper, we proposed a *multi-level evolution framework* designed to separately evaluate **robustness** against semantic-preserving textual perturbations and **memorization** when subtle semantic changes occur. By systematically generating variants of programming tasks—through mutation, paraphrasing, and code-rewriting—and introducing clear, interpretable metrics (such as RR_{mut}, RR_{par}, RAD, Sim, and MRI), we provided a comprehensive approach for distinguishing genuine coding capability from superficial memorization.

Our experiments on both introductory (MBPP+) and more complex real-world tasks (BIGCODEBENCH) produced several key insights. First, we found that **robustness improves significantly as models grow from small to intermediate scales, but plateaus beyond that**, particularly against character-level perturbations. Second, we observed that **paraphrasing generally poses less challenge than mutation, and can sometimes even enhance clarity and improve performance**, showing the importance of evaluating both subtle and substantial linguistic changes. Third, **task complexity significantly influences robustness, with complex, multi-library scenarios demonstrating noticeably lower robustness** compared to simpler problems.

In analyzing fine-tuning strategies, we noted distinct patterns: **SFT substantially improves raw accuracy but at the expense of high memorization risk**, whereas **PPO provides a more balanced trade-off, moderately improving accuracy without significantly increasing memorization**.

Implications. For practitioners aiming to develop reliable code-generation tools, our findings recommend: (1) combining general-purpose pretraining with explicit robustness-focused perturbation methods, (2) favoring reinforcement learning strategies like PPO over pure supervised fine-tuning when true generalization is critical, and (3) routinely reporting memorization metrics, such as MRI, alongside standard accuracy measures.

6 Limitations

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

(1) Mitigation Approach: Further research is needed for reducing the impact of memorization and improving robustness.

(2) Evaluation Transferability: While our current evaluation metrics are tailored for code generation, exploring their applicability to other domains, such as mathematical reasoning, could provide valuable insights.

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

7 Ethical Considerations

Our multi-level evolution framework is guided by ethical principles to ensure responsible outcomes.

(1) Data: Our dataset is constructed from MBPP-Plus and BigCodeBench dataset, which guarantees ethical 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 robustness and memorization in LLM code generation tasks. We encourage the responsible use of the evolved dataset for educational and scientific 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 semantics in writing.

References

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, 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.

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.

Valentin Hartmann, Anshuman Suri, Vincent Bindschaedler, David Evans, Shruti Tople, and Robert West. 2023. Sok: Memorization in general-purpose large language models. *Preprint*, arXiv:2310.18362.

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.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

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*.

Xingyu Lu, Xiaonan Li, Qinyuan Cheng, Kai Ding, Xuanjing Huang, and Xipeng Qiu. 2024. Scaling laws for fact memorization of large language models. *Preprint*, arXiv:2406.15720.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun

Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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.

Laboni Sarker, Mara Downing, Achintya Desai, and Tevfik Bultan. 2024. Syntactic robustness for llm-based code generation. *Preprint*, arXiv:2404.01535.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *Preprint*, arXiv:1707.06347.

Xiaoyin Wang and Dakai Zhu. 2024. Validating llm-generated programs with metamorphic prompt testing. *Preprint*, arXiv:2406.06864.

Zhepeng Wang, Runxue Bao, Yawen Wu, Jackson Taylor, Cao Xiao, Feng Zheng, Weiwen Jiang, Shangqian Gao, and Yanfu Zhang. 2024. Unlocking memorization in large language models with dynamic soft prompting. *Preprint*, arXiv:2409.13853.

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.

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

Appendix

A Prompts for Task Generation

We provide the full instruction prompts used to generate each evolution variant (mutation, paraphrasing, and code-rewriting) with GPT-4.1. For each evolution type, the system and user messages are shown as passed to the API.

A.1 Mutation Evolution

System Prompt

System: You are a helpful assistant. Your goal is to transform a given 'coding task prompt' into a new version. Follow the instructions carefully to transform the prompt.

Mutation Evolution User Prompt

User: Given a coding task description "The Given Prompt" and its canonical solution "Code", perform the following steps:

- X word-scrambling operations
- Y random-capitalization operations
- Z character-noising operations

Definitions (one "operation" = one change):

- **Word scrambling**: choose a single word (alphabetic token) and randomly shuffle its internal letters.
- **Random capitalization**: flip the case of one letter (upper to lower or lower to upper) anywhere in the text.
- **Character noising**: insert, delete, **or** substitute one character (letter, digit, or punctuation).

Please gives your answers to "Mutation Prompt" without any additional text or explanation.

Response: Format your response as:

```
Mutation Prompt:
[Updated task description]
```

NOTE: The values X, Y, and Z — representing the number of word-scrambling, random-capitalization, and character-noising operations respectively — are automatically computed based on the length of the original prompt. Specifically, we apply a total of ≈ 4 noise operations per 5 words. We first ensure at least one operation of each type is included (i.e., X, Y, Z ≥ 1), then randomly distribute the remaining operations among the three types. This strategy ensures a consistent noise budget proportional to the prompt's length while maintaining diversity in corruption types.

A.2 Paraphrasing Evolution

Paraphrasing Evolution User Prompt

User: Given a coding-task description "The Given Prompt", produce a paraphrased version called "Paraphrased Prompt".

Guidelines:

1. Keep the task's meaning, requirements, and input/output specifications identical.
2. Refresh the wording: use synonyms, change sentence order, or rephrase clauses to add light linguistic "noise," but do **not** drop or add information.
3. Preserve any code-related tokens (e.g., variable names, file names, I/O examples) exactly as they appear unless the original prompt explicitly marks them as placeholders.
4. Retain the original structural cues—for example, if the prompt begins with 'Write a Python function…', your rewrite should also begin with that instruction, albeit rephrased

Please gives your answers to "Paraphrased Prompt" without any additional text or explanation. **Response:** Format your response as:

```
Paraphrased Prompt:
[Updated task description]
```

A.3 Code-Rewriting Evolution

Code-Rewriting Evolution User Prompt

User: You are a helpful assistant. Your goal is to transforms a given 'coding task prompt' into a new version. Follow the instructions carefully to transform the prompt. Given a coding task description "The Given Prompt" and its canonical solution "Code", perform the following steps:

1. Modify the canonical solution to create "New Code" by altering its core logic or structure (e.g., changing loop behavior, condition checks, or algorithmic approach).
2. Avoid superficial changes like variable renaming. Ensure the modified code has different semantics so that we expect anyone who can solve the original problem can also solve your rewritten one.
3. Update "The Given Prompt" to create "Rewritten Prompt". The new prompt must:
   (a) - Match the original's input/output parameter format exactly.
   (b) - Reflect the modified code's logic changes explicitly (e.g., 'count elements divisible by 5' instead of 'sum even numbers').
   (c)
4. "New Code" must run in O(n log n) time or better (unless the original algorithm is already asymptotically optimal).
5. Ensure "New Code" runs in O(n log n) time or better (if possible).

Response: Format your response exactly as:

```
New Code:
```

```
[code]

Explanation:
[logic changes]

Rewritten Prompt:
[updated description]

Old Entry Point:
[original function name]

New Entry Point:
[updated function name]
```

Additionally, we ensured the validity of test cases for all rewritten tasks across both datasets. For MBPP+, we reuse the official test case inputs and generate the expected outputs using the rewritten ground-truth solutions, ensuring direct comparability. For BigCodeBench, we adopt the procedure outlined in (Zhuo et al., 2024), constructing test cases for each rewritten task based on their guidelines to guarantee consistency and correctness. We installed all packages required by both dataset for assessing function correctness.

B Examples of Clearer Paraphrased Prompts

Mbpp/604

Original Prompt: Write a function to reverse words separated by spaces in a given string.
Paraphrased Prompt: Create a function that takes a string as input and returns the string with all words, which are divided by spaces, reversed in order.

Mbpp/752

Original Prompt: Write a function to find the nth jacobsthal number. https://www.geeksforgeeks.org/jacobsthal-and-jacobsthal-lucas-numbers/ 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683, 1365, 2731, ...
Paraphrased Prompt: Create a function that computes the nth Jacobsthal number. Refer to https://www.geeksforgeeks.org/jacobsthal-and-jacobsthal-lucas-numbers/ for more information. The sequence begins as follows: 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683, 1365, 2731, ...

Mbpp/753

Original Prompt: Write a function to find minimum k records from tuple list. https://www.geeksforgeeks.org/python-find-minimum-k-records-from-tuple-list/ - in this case a verbatim copy of test cases.
Paraphrased Prompt: Create a function that retrieves the smallest k elements from a list of tuples. Refer to https://www.geeksforgeeks.org/python-find-minimum-k-records-from-tuple-list/ and use the provided test cases exactly as they are.

C Examples of Harmless Internal Logic Modifications

We listed two representative cases from Llama3.1-8B-Instruct where a model has preserved all function and variable names yet altered only the internal implementation (e.g., swapping loops for comprehensions or changing search strategies), illustrating "harmless" syntactic similarity.

C.1 Task Mbpp/18: List Comprehension <-> Explicit Loop

Model Answer in Original Set (list comprehension)

```
def remove_dirty_chars(string, second_string):
    second_string_set = set(second_string)
    filtered = [c for c in string if c not in
     second_string_set]
    return ''.join(filtered)
```

Model Answer in Rewritten Set (explicit loop)

```
def remove_dirty_chars(string, second_string):
    second_string_set = set(second_string)
```

```
    result = []
    for c in reversed(string):
        if c not in second_string_set:
            result.append(c)
    return ''.join(reversed(result))
```

Here, the function name ‘remove_dirty_chars‘, parameters, lookup-set variable, and character variable ‘c‘ all remain identical; only the comprehension is replaced by a two-step loop.

C.2 Task Mbpp/793: Reverse-Index Lookup Binary Search

Model Answer in Original Set (slice + index)

```
def last(arr, x):
    try:
        return len(arr) - 1 - arr[::-1].index(x)
    except ValueError:
        return -1
```

Model Answer in Rewritten Set (binary search)

```
def last(arr, x):
    low, high = 0, len(arr) - 1
    result = -1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            result = mid
            low = mid + 1
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return result
```

Again, ‘last‘, ‘arr‘, and ‘x‘ are preserved; the lookup logic is simply swapped from a reverse-slice search to an iterative binary-search routine.

D Evolved-Task Generation (GPT-4.1)

- **API version**: `gpt-4.1-2025-04-14`.
- **Prompt template**: shown in Appendix. A.
- **Parameters**: temperature 0.7, top-p 1.0, max-tokens 1080.
- **Post-processing**: regex clean-up.