# MEMORIZE OR GENERALIZE? EVALUATING LLM CODE GENERATION WITH CODE REWRITING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large language models (LLMs) have recently demonstrated exceptional code generation capabilities. However, there is a growing debate whether LLMs are mostly doing memorization (i.e., replicating or reusing large parts of their training data) versus generalization (i.e., beyond training data). Existing evaluations largely proxy memorization with surface/structural similarity, thereby conflating benign reuse of repeated code with harmful recall and neglecting task correctness under semantic variation. We define harmful memorization behaviorally as *failure at high similarity* and introduce a semantic perturbation *code rewriting*, which rewrites a semantically different answer at a similar difficulty level for a given coding task, then reverse-engineers a novel coding task. We further propose *Memorization Risk Index (MRI)*, a normalized score that combines two signals: (i) how similar the model's answer for the rewritten task is to the original ground-truth solution, and (ii) how much performance drops from the original task to its rewritten counterpart. MRI is high only when both conditions hold—when the model outputs similar code but fails the perturbed task—thereby capturing harmful memorization rather than benign reuse of repeated code. Empirical evaluations on code generation benchmarks MBPP+ and BIGCODEBENCH reveal that (1) memorization does not increase with larger models and in many cases alleviates as they scale; (2) supervised fine-tuning (SFT) improves accuracy while introduces memorization; (3) reinforcement learning with proximal policy optimization (PPO) achieves a more balanced trade-off between memorization and generalization.

## 1 INTRODUCTION

Large language models (LLMs) have made incredible advances in automated code generation, and are rapidly becoming essential tools in software development (Sourcegraph, 2024; Tabnine, 2024; Team et al., 2023; Anthropic, 2025; Chen et al., 2021). 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. These advancements raise an important question: when do LLMs truly generalize to new programming tasks, and when are they merely reproducing memorized training examples?

Understanding memorization in code generation is critical. Existing evaluations largely measure memorization via surface or structural overlap (e.g., regurgitation audits, contamination filters, and entropy-based detectors) (Yang et al., 2024; Riddell et al., 2024; Dong et al., 2024), treating high similarity as evidence of memorization. This conflates benign reuse of repeated code (i.e. idioms, APIs) with harmful recall and, crucially, does not test whether the model solves the task under semantic variation.

To systematically study harmful memorization, we build on the intuition that performance gaps under semantic perturbations contribute to reveal whether a model is generalizing or harmful memorizing. If a model simply recalls solutions, even small semantic changes could cause large accuracy drops, often accompanied by high overlap with training-like code (Bayat et al., 2024). Concretely, we propose *code-rewriting*, which introduces semantic shifts to prompts while maintaining similar syntax, to investigate whether the success of a model comes from genuine reasoning or harmful memorization. To quantify these behaviors, we introduce Memorization Risk Index (MRI), a nor-

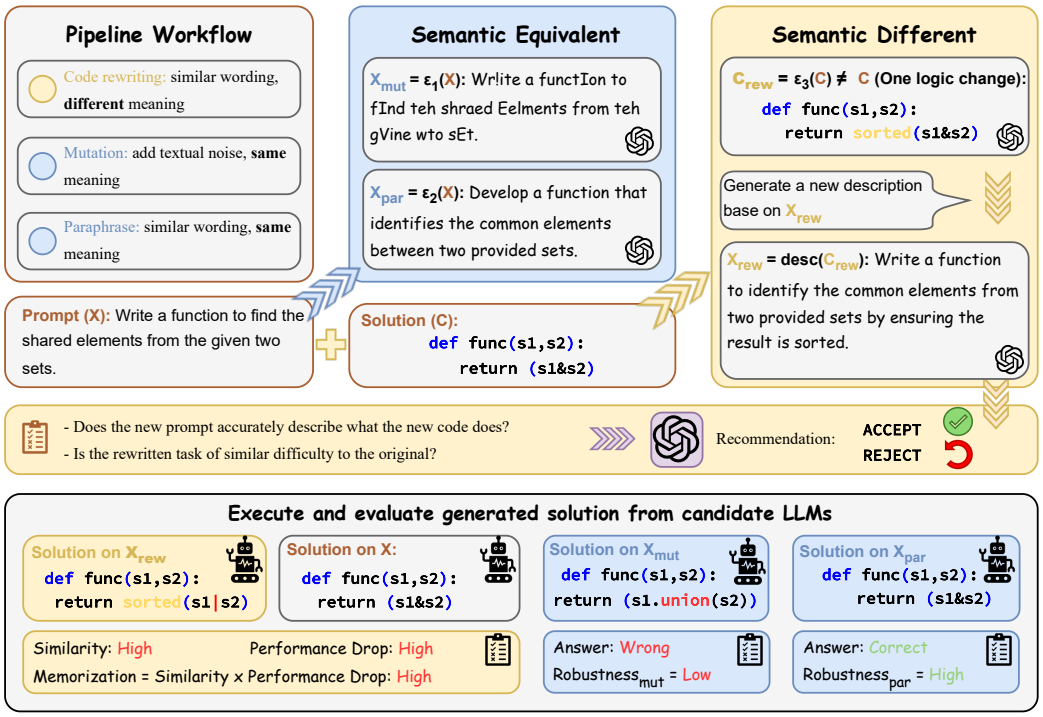



Figure 1: Our proposed Code Rewriting vs. Popular semantic equivalent perturbations. $X$ denotes text and $C$ denotes code. *Code rewriting* that creates semantically different tasks, first rewrite a new code solution $C_{\text{rew}}$ from the origin solution $C$, then generating a new description $X_{\text{rew}}$ based on $C_{\text{rew}}$. A judge agent will then choose to accept or reject the *code rewriting* task for quality assurance. *Mutation* and *paraphrase* that create semantically equivalent tasks, are included for robustness evaluation as a comparison to memorization. All perturbations are performed by GPT-5, shown as the ChatGPT logo. Generation prompts are in Appendix B.

malized score that combines two signals: (i) how similar the model's answer for the *code rewriting* task is to the original ground-truth solution (combining both semantic and syntax level similarity), and (ii) how much performance drops from the original task to its *code rewriting* counterpart. MRI captures harmful memorization as *failure under high similarity* on code-rewriting tasks.

**Terminology.** In this paper, we use the term *memorization* to specifically denote *harmful memorization*: which we define as cases that (1) exhibit high similarity to the original solution and (2) lead to performance drops under semantically altered *code rewriting*. Unless otherwise stated, all subsequent uses of "memorization" follow this definition.

To differentiate our method from existing work in evaluating robustness (Chen et al., 2024; 2023; Mastropaolo et al., 2023; Wang et al., 2022), we also include two semantic-preserving perturbations, *mutation* and *paraphrase*, as reference baselines. We report Relative Accuracy Drop (RAD) to measure LLMs performance consistency under semantic-preserving perturbations. Our primary analysis still targets harmful memorization via the semantics-altering code-rewriting perturbation and MRI.

Our evaluation include coding benchmarks across different difficulty levels, from introductory problems in MBPP+ (Liu et al., 2023) to more difficult tasks in BIGCODEBENCH (Zhuo et al., 2024). Furthermore, we investigate the effect of post-training strategies by comparing Supervised Fine-Tuning (SFT) and Proximal Policy Optimization (PPO). Our results reveal the following trends: **(1)** memorization does not increase with larger models and in many cases improves as they scale; **(2)** memorization alleviates rapidly on simpler tasks but persists on more difficult ones; **(3)** SFT improves raw accuracy but substantially amplifies memorization; **(4)** PPO achieves a more balanced trade-off, mitigating memorization while maintaining competitive accuracy.

In summary, our work makes the following key contributions:

- We propose a novel **automated pipeline** for *code rewriting*, which rewrites a semantically different answer at a similar difficulty level for a given coding question, then reverse-engineers a novel coding question.

- We introduce MRI, a metric that captures harmful memorization as **failure under high similarity** on *code rewriting* tasks, rather than treating similarity alone as memorization.
- We conduct a comprehensive empirical study across benchmarks and training strategies, providing insights into when LLMs memorize in code generation.

## 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), LiveCodeBench (Jain et al., 2024) and 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 memorization behaviors to prompt variations.

### 2.2 Memorization in Code Generation

A model that memorizes may output correct-looking solutions simply because it has seen near-identical problems during pre-training, rather than reasoning about program semantics (Pappu et al., 2024; Duan et al., 2024; Kassem et al., 2024; Carlini et al., 2019; Bayat et al., 2024; Xie et al., 2024). Such behavior can mislead evaluation benchmarks, inflate metrics, and compromise trustworthiness when models are deployed in real-world development environments (Hartmann et al., 2023; Lu et al., 2024; Staab et al., 2023; Zanella-Béguelin et al., 2020).

In code generation, prior work operationalizes general memorization as *regurgitation*—via prefix to suffix extraction, mass sampling with clone detection against the training corpus, and contamination-aware splits of HumanEval/MBPP (Chen et al., 2021)—and under these measurements reports that the measured general memorization rate *increases with model size* (Yang et al., 2024; Al-Kaswan et al., 2024; Wang et al., 2024). However, general memorization is not inherently harmful: if a training-like solution still satisfies the a semantic different question, re-use does not constitute risk (it is correct and passes tests) (Bayat et al., 2024). To distinguish harmful memorization from genuine generalization, we introduce *code-rewriting*, which deliberately shifts task semantics while preserving surface syntax, and we quantify it with a *Memorization Risk Index (MRI)* that multiplies similarity to the original solution by the relative accuracy drop under the semantic shift (high only when the answer copied surface forms but fail on the task with new semantics). Lai et al. (2022) uses semantic perturbations—changing the reference solution's semantics without increasing difficulty—to probe general memorization; unlike their manually authored data-science tasks, we perform automated code rewriting and introduce MRI.

## 3 Methodology

### 3.1 Code Rewriting

*Code rewriting* is used to evaluate a model's memorization via solving **semantically different** problems that are superficially similar to original tasks. The automated pipeline to generate *code rewriting* tasks is shown in Figure 1. Specifically, we first modify **one logic** in 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 while similar to origin tasks in syntax. Formally, let $x \in T$ be the original prompt in text space $T$ and $c \in C$ its ground truth code solution in code space $C$. We apply a rewriting function $\epsilon_3$ 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:

$$x_{rew} = \text{desc}(c_{rew}) \tag{1}$$

$$\text{where} \quad \text{sig}(c_{rew}) = \text{sig}(c), \quad c_{rew} \neq c \tag{2}$$

where desc$(\cdot)$ denotes generating a description from code, and 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.

**Data Validation.** To ensure the reliability of *code rewriting* datasets, we conducted both LLM-as-a-judge and manual quality assurance. For LLM-as-a-judge (shown in Figure 1), we forward *code rewriting* tasks to GPT-5 (OpenAI, 2025) to check (i) if the rewritten code match the rewritten prompt and (ii) if the rewritten task align with the original task in difficulty. For manual validation, two experienced python programmers randomly reviewed 10% of generated evolution problems for all three evolution types to ensure their quality. We also provide 5 regressed tasks for each dataset (PASSED in original but FAILED in *code rewriting*) in Appendix D to show difficulty alignment.

#### 3.1.1 Metric-Memorization Risk Index

MRI consists of two signals: (i) how similar the model's answer for the rewritten task is to the original ground-truth solution, and (ii) how much performance drops from the original task to its rewritten counterpart.

**(i) Similarity.** For every rewritten task $i \in \mathcal{T}_{\text{rew}}$, where $\mathcal{T}$ refers to a task set, we measure two similarities between the model-generated solution for the rewritten version of task $i$ and the ground-truth solution of its original version:

- Semantic level AST similarity: normalized tree-edit overlap between abstract-syntax trees
- Syntax level edit similarity: $(1 - (\text{Levenshtein distance}/\text{max-len})$, capturing token-level overlap.

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

$$S_i = \frac{\text{AST}_i + \text{Edit}_i}{2} \tag{3}$$

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

$$\text{Sim}(\mathcal{T}_{\text{rew}}) = \frac{1}{|\mathcal{T}_{\text{rew}}|} \sum_{i \in \mathcal{T}_{\text{rew}}} S_i, \quad \text{Sim} \in [0, 1]. \tag{4}$$

**(ii) Relative Accuracy Drop for Rewriting.** For a task set $\mathcal{T}$, Pass@1 is reported as $\text{Acc}(\mathcal{T})$. To capture the performance loss induced by semantic rewriting, we define

$$\text{RAD}_{\text{rew}} = \max\left(0, \frac{\text{Acc}(\mathcal{T}_{\text{ori}}) - \text{Acc}(\mathcal{T}_{\text{rew}})}{\text{Acc}(\mathcal{T}_{\text{ori}})}\right), \quad \text{RAD}_{\text{rew}} \in [0, 1]. \tag{5}$$

$\text{RAD}_{\text{rew}} = 0$ when rewriting does not hurt accuracy and increases when it does; the $\max(0, \cdot)$ prevents negative values when the performance on rewritten tasks happen to be better.

**MRI.** Finally, we introduce the MRI, defined as the product of solution-similarity and relative accuracy drop:

$$\text{MRI} = \text{Sim}(\mathcal{T}_{\text{rew}}) \times \text{RAD}_{\text{rew}}, \quad \text{MRI} \in [0, 1]. \tag{6}$$

MRI is high only when both conditions for harmful memorization hold: (i) the model copies the original solution's surface form (high $\text{Sim}(\mathcal{T}_{\text{rew}})$) and (ii) that copied solution now fails (high $\text{RAD}_{\text{rew}}$). This multiplicative design sharply distinguishes memorization from generalization.

### 3.2 Mutation and Paraphrase

To differentiate *code rewriting* from **semantic-preserving** perturbation techniques in work evaluating robustness (Chen et al., 2024; 2023; Mastropaolo et al., 2023; Wang et al., 2022), we include *mutation* and *paraphrase*, as reference baselines. These two perturbations reveal if LLM could generate consistent and correct responses under minor surface level changes (Li et al., 2022). *Mutation* and *paraphrase* are adapted in spirit from ReCode's robustness benchmark (Wang et al., 2022).

**Mutation.** 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. 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 the original semantics:

$$x_{mut} = \epsilon_1(x), \quad x, x_{mut} \in T \tag{7}$$

where $\epsilon_1$ injects textual noise without altering the problem's underlying meaning.

**Paraphrase.** Paraphrase 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:

$$x_{par} = \epsilon_2(x), \quad x, x_{par} \in T \tag{8}$$

where $x_{par}$ is a semantically equivalent but textually different paraphrase of $x$.

#### 3.2.1 METRIC—ROBUSTNESS RELATIVE ACCURACY DROP

Once we perturb a prompt without changing its semantics, what fraction of previously-solved tasks remain solved? To answer this question and differentiate robustness with memorization, for each semantic-preserving transformation $p \in \{\mathrm{mut}, \mathrm{par}\}$ (mutation/paraphrase), we define the **Robustness Relative Accuracy Drop**:

$$\mathrm{RAD}_{\mathrm{p}} = \max\left(0,\ \frac{\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}}) - \mathrm{Acc}(\mathcal{T}_p)}{\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})}\right), \quad \mathrm{RAD}_{\mathrm{p}} \in [0, 1]. \tag{9}$$

Here, $\mathrm{Acc}(\cdot)$ denotes Pass@1 on the indicated task set section 3.1.1. $\mathrm{RAD}_{\mathrm{p}} = 0$ (high robustness) when semantic-preserving changes do not hurt accuracy and increases toward 1 as performance degrades (low robustness).

### 3.3 FINE-TUNING METHODS

To investigate the memorization phenomenon, we use the original tasks in MBPP+ and BIGCODEBENCH for fine-tuning[1]. More training details regarding SFT/RL can be found at Appendix F.

#### 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 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 where model begin to memorize training examples. Therefore, we distinguish between early-stage and late-stage memorization by the checkpoint where the loss on the validation set begins to increase. **We select such checkpoint for evaluation to distinguish memorization from overfitting**.

#### 3.3.2 REINFORCEMENT LEARNING

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 for simplicity, 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. **We evaluate using the checkpoint that achieves the highest validation reward**.

[1]For clarity, both SFT and PPO are initialized from the same base model and trained independently; PPO is not performed on top of an SFT checkpoint.

# 4 EXPERIMENT SETUP

## 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**. Due to the small size of MBPP+ test split ($n = 78$), estimation on this split may be imprecise and directional, we use BIGCODEBENCH to explore the impact of fine-tuning strategies on memorization.

**Task Generation.** For each original task, we generate one perturbed variant for each of *code rewriting, mutation* and *paraphrase*. More about the generation process is given in Appendix G.

## 4.2 MODELS

In this paper, we conduct the scale-up experiments on Qwen-2.5 series (Hui et al., 2024), Qwen-2.5-Coder series (Qwen et al., 2025), Llama-3.1 series (Dubey et al., 2024) and Llama-4 series (AI, 2024). For fine-tuning, we choose Qwen-2.5-7B, and Qwen-2.5-Coder-7B. All training and inference were conducted on a server equipped with 4 NVIDIA A100 GPUs (80GB), with a total computational budget of approximately 40 GPU hours, using PyTorch and HuggingFace Transformers.

# 5 RESULT ANALYSIS

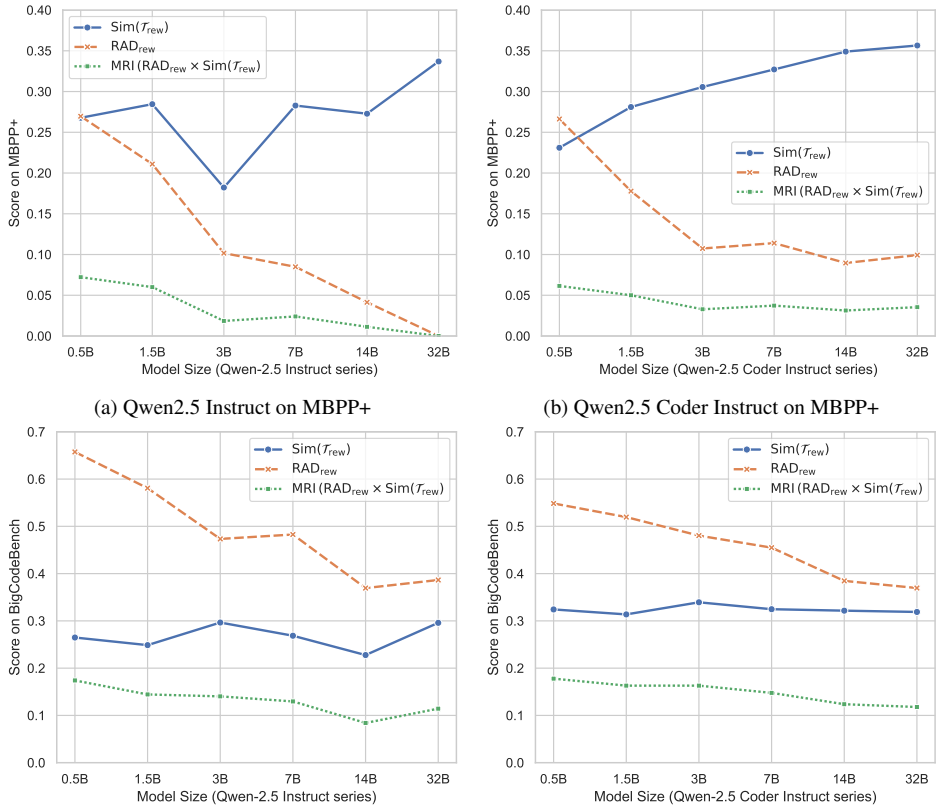
## 5.1 MEMORIZATION ANALYSIS ON INSTRUCT MODELS

**Memorization does not increase with larger models and in many cases decreases as they scale.** Across Qwen2.5 Instruct and its Coder Instruct families, scaling is *associated with* lower $\mathrm{RAD}_{\mathrm{rew}}$ and hence lower MRI. On MBPP+ (see Figure 2a and Figure 2b), Qwen-Instruct's MRI falls from $0.0722$ at `0.5B` to $0.0113$ at `14B`, reaching $0.0000$ at `32B`, driven by a decrease in $\mathrm{RAD}_{\mathrm{rew}}$ from $0.2697 \rightarrow 0.0414 \rightarrow 0.0000$. A similar pattern holds for Qwen-Coder (MRI $0.0615 \rightarrow 0.0313 \rightarrow 0.0354$ as $\mathrm{RAD}_{\mathrm{rew}}$ goes from $0.2663 \rightarrow 0.0896 \rightarrow 0.0993$). Notably, $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ does not uniformly decline with scale (e.g., Qwen-Instruct: $0.2678$ at `0.5B` $\rightarrow$ $0.3369$ at `32B`), indicating that larger models may continue to reuse surface patterns; however, because their failures under semantic shifts largely vanish, such reuse is not harmful and thus produces much lower MRI.

| Model | MBPP+ | | | BigCodeBench | | |
|---|---|---|---|---|---|---|
| | $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ (↓) | $\mathrm{RAD}_{\mathrm{rew}}$ (↓) | **MRI** (↓) | $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ (↓) | $\mathrm{RAD}_{\mathrm{rew}}$ (↓) | **MRI** (↓) |
| Llama-3.1-8B-Instruct | 0.1486 | 0.0133 | 0.0020 | 0.2132 | 0.4444 | 0.0947 |
| Llama-3.1-70B-Instruct | 0.1518 | 0.0000 | 0.0000 | 0.2404 | 0.3676 | 0.0884 |
| *Llama-3.1-Instruct Series (mean)* | 0.1502 | 0.0067 | 0.0010 | 0.2268 | 0.4060 | 0.0916 |
| Llama-4-Scout-17B-Instruct (16E) | 0.1446 | 0.0160 | 0.0023 | 0.2343 | 0.3909 | 0.0916 |
| Llama-4-Maverick-17B-Instruct (128E) | 0.2669 | 0.0307 | 0.0082 | 0.2357 | 0.3953 | 0.0932 |
| *Llama-4-Instruct Series (mean)* | 0.2057 | 0.0234 | 0.0053 | 0.2350 | 0.3931 | 0.0924 |

Table 1: Memorization risk for Llama-3.1 and Llama-4 instruct models. MRI persists in harder tasks (BIGCODEBENCH), as $\mathrm{RAD}_{\mathrm{rew}}$ stays high even as $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ is comparable.

On BIGCODEBENCH (see Figure 2c and Figure 2d), the effect from scaling up is milder and sometimes non-monotonic. Qwen-Instruct's MRI drops from $0.1740$ (`0.5B`) to $0.0841$ (`14B`) but increases to $0.1143$ at `32B`, with $\mathrm{RAD}_{\mathrm{rew}}$ trending from $0.6574 \rightarrow 0.3694 \rightarrow 0.3865$. On the other hand, Qwen-Coder shows a steadier decline ($0.1778 \rightarrow 0.1178$ from `0.5B`$\rightarrow$`32B`) with relatively flat $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$. Overall, scale reduces memorization primarily by improving resistance to semantic




(a) Qwen2.5 Instruct on MBPP+

(b) Qwen2.5 Coder Instruct on MBPP+

(c) Qwen2.5 Instruct on BIGCODEBENCH

(d) Qwen2.5 Coder Instruct on BIGCODEBENCH

Figure 2: Scaling trends in MRI across Qwen-2.5 Instruct vs. Coder on MBPP+ and BIGCODEBENCH.

shifts ($\text{RAD}_{\text{rew}}$), while surface-form similarity can remain high. The gains are pronounced on simpler tasks (MBPP+) and partially eroded on harder ones (BIGCODEBENCH); on BIGCODEBENCH, the non-zero MRI is explained by persistently high $\text{RAD}_{\text{rew}}$ with roughly unchanged $\text{Sim}(\mathcal{T}_{\text{rew}})$.

We also evaluate on Llama families (see Table 1). While Llama 3.1 exhibit similarly low MRI as scale increases, Llama 4 [2] shows comparable MRI on both dataset. On MBPP+, the MRI from Llama-3.1 (`8B`/`70B`) declined in small degree ($0.0020 \rightarrow 0.0000$), and Llama-4 models are near zero ($0.0023$ and $0.0082$); on BIGCODEBENCH, Llama-3.1 shifts little ($0.0947 \rightarrow 0.0884$), and Llama-4 remains comparably low but non-zero ($0.0932$ and $0.0916$). These results reveal a task-dependency on harmful memorization: on easier problems, larger Llama models effectively drive $\text{RAD}_{\text{rew}} \rightarrow 0$ (hence negligible MRI) even when $\text{Sim}(\mathcal{T}_{\text{rew}})$ is moderate, whereas on BIGCODEBENCH the non-zero risk is dominated by persistent $\text{RAD}_{\text{rew}} = 0.4060$ for Llama 3.1 series and $0.3931$ for Llama 4 series at similar similarity levels.

### 5.1.1 ADDITIONAL FINDINGS

**Memorization declines rapidly on simpler tasks but persists on more difficult ones.** On the introductory-level tasks in MBPP+ (see Figure 2a and Figure 2b), memorization risk (MRI) decreases notably as models scale up. For instance, for Qwen-2.5-Instruct's MRI falls from 0.0722 at `0.5B` parameters to effectively zero at `32B`. Conversely, on the more challenging BIGCODEBENCH (see Figure 2c and Figure 2d), MRI values remain significant even at large scales (0.1178 for Qwen-2.5-32B-Instruct). This discrepancy shows that while larger models better capture underlying semantics changes, they do not completely eliminate memorization, especially in scenarios of challenging tasks that demand deeper reasoning.

[2] The two Llama-4 variants we evaluate are MoE models with similar per-token activated compute; their difference is mainly *capacity* (number of experts) rather than dense compute scaling.

**Coder models encourages code reuse but does not substantially increase memorization.** Coder models yield higher $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ than their instruction-only counterparts. For instance, on BIGCODEBENCH, Qwen-2.5-Coder series (Figure 2d) scores $\mathbf{0.3237} \pm 0.0086$ vs. $\mathbf{0.2670} \pm 0.0268$ for the instruction-only variant (Figure 2c) (mean $\pm$ SD over 6 seeds). However, $\mathrm{RAD}_{\mathrm{rew}}$ remains comparable across these variants, translating to only a slight increase in MRI ($\mathbf{0.1367} \pm 0.0224$ vs. $\mathbf{0.1142} \pm 0.0247$, mean $\pm$ SD over 6 seeds). This pattern suggests code-focused pre-training promotes superficial reuse of training data without significantly increase harmful memorization.

### 5.2 IMPACT OF FINE-TUNING STRATEGIES ON MEMORIZATION

Figure 3 shows notable differences in memorization across different fine-tuning strategies on Qwen-2.5-7B and Qwen-2.5-Coder-7B on BIGCODEBENCH.

**SFT improves accuracy but introduces high memorization risk.** Models fine-tuned via SFT consistently achieve accuracy gains on original tasks. For Qwen-2.5-7B-SFT, accuracy was boosted from $0.3158 \rightarrow 0.3772$ on BIGCODEBENCH and increasing from $0.3684 \rightarrow 0.4079$ on the coder counterpart. However, for both Qwen-2.5-7B-SFT and Qwen-2.5-Coder-7B-SFT, these improvements come with significant increases in memorization, as indicated by much higher MRI scores (e.g. $0.0799 \rightarrow 0.1747$ for Qwen-2.5-7B-SFT and $0.1392 \rightarrow 0.1921$ on for Qwen-2.5-Coder-7B-SFT). These trends reveals that SFT enhances surface-level accuracy at the expense of genuine generalization.

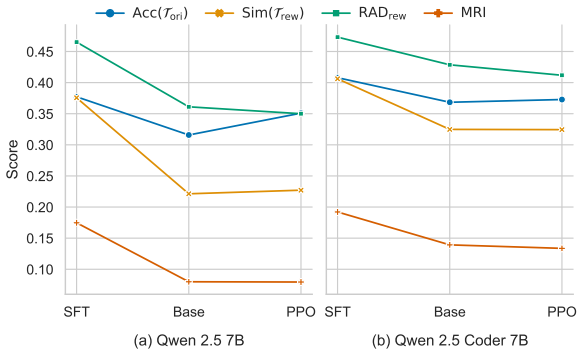



Figure 3: Effect of fine-tuning on Qwen-2.5-7B (base and Coder) on BIGCODEBENCH. SFT raises $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$ but also increases $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ and $\mathrm{RAD}_{\mathrm{rew}}$, inflating MRI; PPO preserves or modestly improves accuracy while keeping $\mathrm{RAD}_{\mathrm{rew}}$ low, yielding a better risk–accuracy trade-off. Checkpoints selected for SFT and PPO follows rules in subsection 3.3. Dataset statistics can be found in subsection 4.1

**PPO balances accuracy improvements and memorization risk.** Across both variants, PPO preserves baseline-level or higher accuracy while sharply reducing memorization risk relative to SFT. On Qwen-2.5-7B, accuracy moves from $0.3158 \rightarrow 0.3509$ (PPO) vs $0.3772$ (SFT), with MRI $0.0799 \rightarrow 0.0795$ (PPO) vs $0.1747$ (SFT); Similar trend was revaled by Qwen-2.5-Coder-7B, where accuracy is $0.3684 \rightarrow 0.3728$ (PPO) vs $0.4079$ (SFT), with MRI $0.1392 \rightarrow 0.1336$ (PPO) vs $0.1921$ (SFT). Overall, PPO yields a better risk–accuracy trade-off by keeping MRI near or below base levels while offering milder accuracy gains, in contrast to SFT's larger accuracy improvements accompanied by substantially higher MRI.

**Implications for Fine-Tuning Decisions.** The choice between SFT and reinforcement-based approaches such as PPO is ultimately determined by how one prioritizes the trade-off between accuracy and memorization risk. If maximizing accuracy is the priority and the risks associated with memorization are acceptable, then SFT remains the optimal strategy. However, in settings where generalization and minimizing memorization risk are critical, PPO provides a better balance by offering modest accuracy improvements while considerably reducing memorization.

### 5.3 ROBUSTNESS TO SEMANTIC-PRESERVING PERTURBATIONS

We differentiate from memorization by using two semantic-preserving perturbations—*mutation* and *paraphrase*—as reference baselines, and we quantify consistency under these baselines with RAD; our primary analysis remains memorization via semantics-altering rewriting and MRI.

**Mutation remains more challenging** Across BIGCODEBENCH, *mutation* induces a moderate RAD while *paraphrase* exhibits a milder influence on model accuracy: averaged over all models, $\mathrm{RAD}_{\mathrm{mut}} = 0.20 \pm 0.12$ and $\mathrm{RAD}_{\mathrm{par}} = 0.06 \pm 0.04$, compared to a much larger semantics-altering rewriting drop of $\mathrm{RAD}_{\mathrm{rew}} = 0.46 \pm 0.09$. On MBPP+, both mutation and rewriting are modest

($\mathrm{RAD}_{\mathrm{mut}} = 0.10 \pm 0.08$, $\mathrm{RAD}_{\mathrm{rew}} = 0.10 \pm 0.09$), and paraphrase is essentially invariant ($\mathrm{RAD}_{\mathrm{par}} = 0.01 \pm 0.01$). These results confirm that our primary memorization analysis (via rewriting and MRI) targets a qualitatively different—and much stronger—source of variance than same-semantics perturbations.

**Scaling helps robustness to mutation; coder models show higher sensitivity to mutation on harder tasks.** Mutation accuracy drop decreases with model size on both benchmarks, while paraphrase remains near-zero with small fluctuations at the high end. On BIGCODEBENCH, coder models are the most mutation-sensitive (e.g., Qwen-2.5-coder avg. $\mathrm{RAD}_{\mathrm{mut}} = 0.25 \pm 0.12$) versus their instruction counterparts ($0.20 \pm 0.13$), with Llama families lower still (Llama-3.1 $\mathrm{RAD}_{\mathrm{mut}} = 0.1050$, Llama-4 $\mathrm{RAD}_{\mathrm{mut}} = 0.1206$). On MBPP+, absolute drops are smaller for all families; Llama-4 shows the lowest RAD under mutation ($\mathrm{RAD}_{\mathrm{mut}} = 0.0578$). Paraphrase occasionally yields zero or even negative drops (i.e., accuracy improves), consistent with minor wording changes sometimes helping the model parse constraints.

| Model | MBPP+ | | | | BigCodeBench | | | |
|---|---|---|---|---|---|---|---|---|
| | $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$ (↑) | $\mathrm{RAD}_{\mathrm{mut}}$ (↓) | $\mathrm{RAD}_{\mathrm{par}}$ (↓) | $\mathrm{RAD}_{\mathrm{rew}}$ (↓) | $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$ (↑) | $\mathrm{RAD}_{\mathrm{mut}}$ (↓) | $\mathrm{RAD}_{\mathrm{par}}$ (↓) | $\mathrm{RAD}_{\mathrm{rew}}$ (↓) |
| Qwen-2.5 0.5B-Instruct | 0.4021 | 0.2763 | 0.0000 | 0.2697 | 0.0947 | 0.4352 | 0.0000 | 0.6574 |
| Qwen-2.5-1.5B-Instruct | 0.5767 | 0.2248 | 0.0000 | 0.2110 | 0.2281 | 0.2115 | 0.0000 | 0.5808 |
| Qwen-2.5-3B-Instruct | 0.6243 | 0.1144 | 0.0000 | 0.1017 | 0.3132 | 0.1989 | 0.0448 | 0.4734 |
| Qwen-2.5-7B-Instruct | 0.6852 | 0.0463 | 0.0000 | 0.0849 | 0.3798 | 0.1409 | 0.0878 | 0.4827 |
| Qwen-2.5-14B-Instruct | 0.7037 | 0.0338 | 0.0000 | 0.0414 | 0.3895 | 0.0631 | 0.0608 | 0.3694 |
| Qwen-2.5-32B-Instruct | 0.7513 | 0.0106 | 0.0000 | 0.0000 | 0.4404 | 0.1474 | 0.0757 | 0.3865 |
| *Qwen-2.5-Instruct (mean ± SD)* | $0.62 \pm 0.12$ | $0.12 \pm 0.11$ | $0.00 \pm 0.00$ | $0.12 \pm 0.10$ | $0.31 \pm 0.13$ | $0.20 \pm 0.13$ | $0.04 \pm 0.04$ | $0.49 \pm 0.11$ |
| Qwen-2.5-coder-0.5B-Instruct | 0.4471 | 0.2367 | 0.0000 | 0.2663 | 0.1088 | 0.4677 | 0.0000 | 0.5484 |
| Qwen-2.5-coder-1.5B-Instruct | 0.5952 | 0.1378 | 0.0000 | 0.1778 | 0.2465 | 0.2954 | 0.0391 | 0.5196 |
| Qwen-2.5-coder-3B-Instruct | 0.6402 | 0.0909 | 0.0000 | 0.1074 | 0.3579 | 0.2304 | 0.0686 | 0.4804 |
| Qwen-2.5-coder-7B-Instruct | 0.7196 | 0.0662 | 0.0294 | 0.1140 | 0.4088 | 0.1803 | 0.1073 | 0.4549 |
| Qwen-2.5-coder-14B-Instruct | 0.7381 | 0.0394 | 0.0143 | 0.0896 | 0.4675 | 0.1463 | 0.0938 | 0.3846 |
| Qwen-2.5-coder-32B-Instruct | 0.7725 | 0.0171 | 0.0000 | 0.0993 | 0.4772 | 0.1857 | 0.1085 | 0.3695 |
| *Qwen-2.5-Coder-Instruct (mean ± SD)* | $0.65 \pm 0.12$ | $0.10 \pm 0.08$ | $0.01 \pm 0.01$ | $0.14 \pm 0.07$ | $0.34 \pm 0.14$ | $0.25 \pm 0.12$ | $0.07 \pm 0.04$ | $0.46 \pm 0.07$ |
| Llama-3.1-8B-Instruct | 0.5529 | 0.1340 | 0.0000 | 0.0133 | 0.3079 | 0.1595 | 0.0513 | 0.4444 |
| Llama-3.1-70B-Instruct | 0.6984 | 0.0795 | 0.0189 | 0.0000 | 0.4175 | 0.0504 | 0.0399 | 0.3676 |
| *Llama-3.1-Instruct (mean)* | 0.6257 | 0.1068 | 0.0095 | 0.0067 | 0.3627 | 0.1050 | 0.0456 | 0.4060 |
| Llama-4-Scout-17B-Instruct (16E) | 0.6614 | 0.0200 | 0.0040 | 0.0160 | 0.4061 | 0.1058 | 0.0670 | 0.3909 |
| Llama-4-Maverick-17B-Instruct (128E) | 0.7751 | 0.0956 | 0.0375 | 0.0307 | 0.4860 | 0.1354 | 0.1119 | 0.3953 |
| *Llama-4-Instruct (mean)* | 0.7183 | 0.0578 | 0.0208 | 0.0234 | 0.4461 | 0.1206 | 0.0894 | 0.3931 |
| **All models (mean ± SD)** | $0.65 \pm 0.11$ | $0.10 \pm 0.08$ | $0.01 \pm 0.01$ | $0.10 \pm 0.09$ | $0.35 \pm 0.12$ | $0.20 \pm 0.12$ | $0.06 \pm 0.04$ | $0.46 \pm 0.09$ |

Table 2: Robustness under semantic-preserving *mutation* and *paraphrase* versus semantics-different rewriting. Mutation induces moderate drops; paraphrase is nearly invariant; rewrites are most disruptive—especially on BIGCODEBENCH—suggesting harmful memorization beyond surface-level robustness. The final row reports column-wise unweighted mean ± sample SD across 16 models.[3]

## 6 CONCLUSION AND FUTURE WORKS

In this paper, we reframed memorization in code generation as (1) exhibit high similarity to the golden solution of original tasks and (2) lead to performance drops under semantically modified variants. We measured such memorization with *code rewriting*—which preserves surface form while changing task semantics—and the Memorization Risk Index (MRI) that multiplies solution similarity with the relative accuracy drop (RAD) under rewriting. This design isolates harmful memorization from benign reuse. Our experiments on MBPP+ and BIGCODEBENCH show: (i) harmful memorization generally decreases with model scale on simpler tasks, (ii) persists more on harder tasks, and (iii) SFT raises accuracy but inflates MRI, while PPO delivers a better risk–accuracy trade-off. Taken together, these findings clarify when errors stem from harmful memorization rather than generalization and motivate the following next steps: (a) mitigation approach: further research is needed for reducing the impact of memorization. (b) 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.

[3]Mean is the unweighted arithmetic average computed *per column* across models; SD is the sample standard deviation (unbiased, $n-1$ denominator). Values are rounded to two decimals.

## ETHICS STATEMENT

Our *code rewriting*, *mutation* and *paraphrase* pipeline is guided by ethical principles to ensure responsible outcomes.

(1) Data: Our dataset is constructed from MBPP+ and BIGCODEBENCH dataset, which guarantees ethical fairness. We actively work to eliminate any harmful or offensive content from the *code rewriting*, *mutation* and *paraphrase* variant datasets to mitigate potential risks.

(2) Responsible Usage and License: The use of the *code rewriting*, *mutation* and *paraphrase* variant datasets is intended solely for evaluating memorization in LLM code generation tasks. We encourage the responsible use of those datasets for educational and scientific purposes, while strongly discouraging any harmful or malicious activities.

## REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our work, we have illustrated the experiment details in the appendix, such as task generation prompts in Appendix B, training details in Appendix F and evolved-task generation configurations in Appendix G. For the dataset and code repository, all evolved tasks and the prompts used during generation will be released publicly upon publication, ensuring reproducibility and facilitating future research.

## REFERENCES

Meta AI. Introducing llama 4: Advancing multimodal intelligence, 2024. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pp. 1–12. ACM, April 2024. doi: 10.1145/3597503.3639133. URL http://dx.doi.org/10.1145/3597503.3639133.

Anthropic. Claude 3.7 sonnet and claude code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025. Accessed: 2025-09-18.

Reza Bayat, Mohammad Pezeshki, Elvis Dohmatob, David Lopez-Paz, and Pascal Vincent. The pitfalls of memorization: When memorization hurts generalization, 2024. URL https://arxiv.org/abs/2412.07684.

Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks, 2019. URL https://arxiv.org/abs/1802.08232.

Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. Nlperturbator: Studying the robustness of code llms to natural language variations, 2024. URL https://arxiv.org/abs/2406.19783.

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

Nuo Chen, Qiushi Sun, Jianing Wang, Ming Gao, Xiaoli Li, and Xiang Li. Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association*

*for Computational Linguistics: EMNLP 2023*, pp. 14857–14873, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.991. URL https://aclanthology.org/2023.findings-emnlp.991/.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or memorization: Data contamination and trustworthy evaluation for large language models, 2024. URL https://arxiv.org/abs/2402.15938.

Sunny Duan, Mikail Khona, Abhiram Iyer, Rylan Schaeffer, and Ila R Fiete. Uncovering latent memories: Assessing data leakage and memorization patterns in large language models. *arXiv preprint arXiv:2406.14549*, 2024.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

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. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

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

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

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

Aly M Kassem, Omar Mahmoud, Niloofar Mireshghallah, Hyunwoo Kim, Yulia Tsvetkov, Yejin Choi, Sherif Saad, and Santu Rana. Alpaca against vicuna: Using llms to uncover memorization of llms. *arXiv preprint arXiv:2403.04801*, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022. URL https://arxiv.org/abs/2211.11501.

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. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Xingyu Lu, Xiaonan Li, Qinyuan Cheng, Kai Ding, Xuanjing Huang, and Xipeng Qiu. Scaling laws for fact memorization of large language models, 2024. URL https://arxiv.org/abs/2406.15720.

Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on github copilot, 2023. URL https://arxiv.org/abs/2302.00438.

OpenAI. Gpt-5. https://openai.com, 2025. Large Language Model.

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. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Aneesh Pappu, Billy Porter, Ilia Shumailov, and Jamie Hayes. Measuring memorization in rlhf for code completion, 2024. URL https://arxiv.org/abs/2406.11715.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang,

Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL https://arxiv.org/abs/2305.18290.

Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models, 2024. URL https://arxiv.org/abs/2403.04811.

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. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Sourcegraph. Sourcegraph cody. https://sourcegraph.com/cody, 2024. Accessed: 2024-05-06.

Robin Staab, Mark Vero, Mislav Balunović, and Martin Vechev. Beyond memorization: Violating privacy via inference with large language models. *arXiv preprint arXiv:2310.07298*, 2023.

Tabnine. Tabnine. https://www.tabnine.com, 2024. Accessed: 2024-05-06.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Shiqi Wang, Li Zheng, Haifeng Qian, Chenghao Yang, Zijian Wang, Varun Kumar, Mingyue Shang, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. Recode: Robustness evaluation of code generation models. 2022. doi: 10.48550/arXiv.2212.10264. URL https://arxiv.org/abs/2212.10264.

Zhepeng Wang, Runxue Bao, Yawen Wu, Jackson Taylor, Cao Xiao, Feng Zheng, Weiwen Jiang, Shangqian Gao, and Yanfu Zhang. Unlocking memorization in large language models with dynamic soft prompting, 2024. URL https://arxiv.org/abs/2409.13853.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. On memorization of large language models in logical reasoning, 2024. URL https://arxiv.org/abs/2410.23123.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pp. 1–13. ACM, April 2024. doi: 10.1145/3597503.3639074. URL http://dx.doi.org/10.1145/3597503.3639074.

Santiago Zanella-Béguelin, Lukas Wutschitz, Shruti Tople, Victor Rühle, Andrew Paverd, Olga Ohrimenko, Boris Köpf, and Marc Brockschmidt. Analyzing information leakage of updates to natural language models. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pp. 363–375, 2020.

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

# APPENDIX

## A USE OF LARGE LANGUAGE MODELS (LLMS)

We made limited use of a large language model (OpenAI's GPT-5) during the preparation of this work. Specifically:

- **Task Generation:** GPT-5 was employed to assist in generating tasks for *code rewriting*, *mutation*, and *paraphrase*. The role of the LLM in this context was restricted to providing task generation; all methodological design, filtering, and integration into our pipeline were carried out by the authors.
- **Writing Assistance:** GPT-5 was additionally used as a language aid for correcting grammar and improving clarity in the writing of the manuscript. The substantive content, research ideas, technical contributions, and overall narrative were conceived and written by the authors without reliance on the LLM.

Beyond these two use cases, no part of the research design, analysis, or interpretation depended on LLM assistance.

## B PROMPTS FOR TASK GENERATION

We provide the full instruction prompts used to generate each evolution variant (mutation, paraphrasing, and code-rewriting) with GPT-5. For each evolution type, the system and user messages are shown as passed to the API.

### B.1 CODE-REWRITING EVOLUTION

System Prompt

**System:** You are an experienced python programmer. Your goal is to transforms a given 'coding task prompt' into a new version. Follow the instructions carefully to transform the prompt.

Code-Rewriting Evolution User Prompt

**User:**

```
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 only ***ONE*** core
    logic or structure. Do not add additional 'if statements' to the code. Avoid
    superficial changes like variable renaming. Ensure the modified code has different
   semantics in a way that ***expected difficulty equivalent to the original problem***.
    Write a #New Entry Point# to the updated code. This function name must be very
    similar or the same as the old entry point, and reflect the modified code's logic
    changes if using #Old Entry Point# could mislead the programmer on the
    #Rewritten Prompt#.
2. Update #The Given Prompt# to create #Rewritten Prompt#. The new prompt must:"
    - Match the original's ***input signature*** exactly, but the output
        format could be different a little bit.
    - Reflect the modified code's logic changes explicitly.
        Retain the original phrasing structure and ***avoid unnecessary rephrasing***
        in a way that the #Rewritten Prompt# syntactically very similar
        to the #The Given Prompt#.
3. If any mismatch arises between new code and new prompt, revise either one
    (without adding more changes) so all constraints in Steps 1-2 are simultaneously
    satisfied.


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]
```

### B.2 Code-Rewriting Evolution LLM Judge

**System Prompt**

**System:** You are an expert code reviewer. Your task is to evaluate whether an evolved coding task maintains appropriate quality standards in terms of prompt-code alignment and difficulty equivalence.

**Code-Rewriting Evolution LLM Judging Prompt**

**User:** Please evaluate the quality of this evolved coding task by analyzing two key aspects:

```
**Original Task:**
Prompt: {original_prompt}
Code: {original_code}
**Evolved Task:**
Prompt: {rewritten_prompt}
Code: {rewritten_code}

**Evaluation Criteria:**
1. **Prompt-Code Alignment**: Does the new prompt accurately describe what the new code
    does?
   - Are the input/output specifications consistent?
   - Does the prompt clearly communicate the expected behavior?
   - Are there any ambiguities or mismatches?

2. **Difficulty Equivalence**: Is the evolved task of similar difficulty to the original?
   - Does it require similar algorithmic thinking?
   - Is the complexity level maintained (not significantly easier or harder)?
   - Does it test similar programming concepts and skills?

**Response Format:**
Provide your evaluation in the following format:

Alignment Score: [1-5, where 5 = perfect alignment, 1 = major misalignment]
Alignment Reasoning: [Brief explanation of why the prompt and code align or don't align]

Difficulty Score: [1-5, where 5 = equivalent difficulty, 3 = acceptable variation, 1 =
    significantly different]
Difficulty Reasoning: [Brief explanation of difficulty comparison]

Overall Recommendation: [ACCEPT/REJECT]
Overall Reasoning: [Brief summary of your decision]
Please be thorough but concise in your evaluation.
```

### B.3 MUTATION EVOLUTION

**System Prompt**

**System:** You are an experienced python programmer. Your goal is to transforms 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 $\approx 4$ noise operations per 5 words. We first ensure at least one operation of each type is included (i.e., X, Y, $Z \geq 1$), 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.

### B.4 PARAPHRASING EVOLUTION

**System Prompt**

**System:** You are an experienced python programmer. Your goal is to transforms a given 'coding task prompt' into a new version. Follow the instructions carefully to transform the prompt.

**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]
```

Additionally, we ensured the validity of test cases for all rewritten tasks across both datasets, and validate each rewritten solution by making it pass its corresponding rewritten unit test. 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.

## C 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.

## D Examples of Regressed Tasks

We randomly selected 5 tasks from each of MBPP+ and BigCodeBench that PASSED in original but FAILED in code_rewriting from the evaluation results in Qwen2.5-Coder-32B-Instruct. For each task, we provide

- Original task prompt and its canonical solution
- Code_rewriting task prompt and the rewritten canonical solution
- Alignment and Difficulty analysis from GPT-5 to investigate (1) if the rewritten prompt aligns with its rewritten solution; (2) whether the difficulty of rewritten task align with its original version.

The following case studies confirms that such performance regression is not caused by the higher difficulty on rewritten tasks.

```
MBPP+, original, TASK 99
"""
Write a function to convert the given
decimal number to its binary equivalent,
represented as a string with no leading
zeros.
assert decimal_to_binary(8) == '1000'
"""

def decimal_to_binary(n):
    return bin(n).replace("0b","")
```

```
MBPP+, code_rewriting, TASK 99
"""
Write a function to convert the given
decimal number to its binary equivalent,
represented as a string in reverse bit
order (least significant bit first).
assert decimal_to_binary_rev(8) == '0001'
"""

def decimal_to_binary_rev(n):
    return bin(n)[2:][::-1]
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The prompt matches the code's behavior: it asks for the binary string with bits reversed (LSB first), and bin(n)
[2:][::-1] does exactly that. Input/output types are consistent (int → str). The expected example 8 → '0001' aligns with the
behavior. Minor optional clarifications (e.g., assume non-negative integers; reversed string may begin with '0') don't affect
alignment.

Difficulty: Both versions rely on bin() and simple string operations (slice/remove prefix vs. slice+reverse). They require
similar reasoning and have comparable complexity.
"""
```

Figure 4: Example of Task-99 from MBPP+ generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

```
MBPP+, original, TASK 224
"""
Write a python function to count the number
of set bits (binary digits with value 1) in
a given number.
assert count_Set_Bits(2) == 1
"""

def count_Set_Bits(n):
    return bin(n)[2:].count('1')
```

```
MBPP+, code_rewriting, TASK 224
"""
Write a python function to count the number
of unset bits (binary digits with value 0)
in a given number's binary representation
(without leading zeros).
assert count_Unset_Bits(2) == 1
"""

def count_Unset_Bits(n):
    return bin(n)[2:].count('0')
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The code counts '0's in bin(n)[2:], which matches "unset bits … without leading zeros" for non-negative integers
(e.g., 2 -> '10' → 1). The sample assertion aligns. Also, some contexts define "unset bits" relative to a fixed width (e.g.,
32-bit), but the prompt correctly narrows this by saying "without leading zeros.

Difficulty: Counting zeros instead of ones requires the same approach, complexity (O(k) over bit-length), and Python
knowledge (bin, count). It tests identical concepts with no added complexity.
"""
```

Figure 5: Example of Task-224 from MBPP+ generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

```
MBPP+, original, TASK 284
"""
Write a function that takes in a list and
element and checks whether all items in the
list are equal to the given element.
assert check_element(["green", "orange",
"black", "white"],'blue')==False
"""

def check_element(list1, element):
  return all(v == element for v in list1)
```

```
MBPP+, code_rewriting, TASK 284
"""
Write a function that takes in a list and
element and checks whether all items in the
list are equal to the given element when
compared case-insensitively (after
converting both to lowercase strings).
assert check_element_ci(["green", "orange",
"black", "white"],'blue')==False
"""

def check_element_ci(list1, element):
  target = str(element).lower()
  return all(str(v).lower() == target for v
in list1)
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The prompt explicitly states to compare "case-insensitively (after converting both to lowercase strings)." The
code converts both the target and each list item via str(...).lower() and uses all(...)—exactly matching the described
behavior. No ambiguity remains about non-string inputs because the prompt specifies conversion to strings.

Difficulty: Algorithmic structure is unchanged (a simple all check over a comprehension). The evolved version adds a minor
preprocessing step (str(...).lower()), which doesn't meaningfully change complexity or required concepts.
"""
```

Figure 6: Example of Task-284 from MBPP+ generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

```
MBPP+, original, TASK 767
"""
Write a python function to count the number
of pairs whose sum is equal to 'sum'. The
funtion gets as input a list of numbers and
the sum,
assert get_pairs_count([1,1,1,1],2) == 6
"""

def get_pairs_count(arr, sum_):
    cnt = 0
    for n in arr:
        cnt += arr.count(sum_ - n)
        if sum_ - n == n:
            cnt -= 1
    return cnt / 2
```

```
MBPP+, code_rewriting, TASK 767
"""
Write a python function to count the number
of ordered pairs (i, j) with i != j whose
sum is equal to 'sum'. The funtion gets as
input a list of numbers and the sum,
assert get_pairs_count_ordered([1,1,1,1],2)
== 12
"""
def get_pairs_count_ordered(arr, sum_):
    cnt = 0
    for n in arr:
        cnt += arr.count(sum_ - n)
        if sum_ - n == n:
            cnt -= 1
    return cnt
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The evolved prompt explicitly asks for ordered pairs (i, j) with i ≠ j summing to the target. The code counts, for
each element, the occurrences of its complement and subtracts one when the element equals its complement—precisely excluding
(i, i). It does not divide by 2, so each (i, j) and (j, i) are both counted. The example assert
get_pairs_count_ordered([1,1,1,1],2) == 12 matches this behavior.

Difficulty: Both tasks use the same O(n²) counting approach and require the same conceptual insight (complement counting and
handling self-pairs). The evolved task tests essentially the same skills and complexity.
"""
```

Figure 7: Example of Task-767 from MBPP+ generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

```
MBPP+, original, TASK 279
"""
Write a function to find the nth decagonal
number.
assert is_num_decagonal(3) == 27
"""

def is_num_decagonal(n):
    return 4 * n * n - 3 * n
```

```
MBPP+, code_rewriting, TASK 279
"""
Write a function to find the nth
dodecagonal number.
assert is_num_dodecagonal(3) == 33
"""

def is_num_dodecagonal(n):
    return 5 * n * n - 4 * n
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The prompt ("find the nth dodecagonal number") matches the code's formula 5n^2 - 4n and the test
is_num_dodecagonal(3) == 33.

Difficulty: Computing dodecagonal vs. decagonal numbers uses the same polygonal-number template P_k(n)=\frac{(k-2)n^2-(k-4)n}
{2}. The evolved task requires the same level of formula application and implementation effort as the original.
"""
```

Figure 8: Example of Task-279 from MBPP+ generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

```
BigCodeBench, original, TASK BigCodeBench/1134
"""
Computes the MD5 hash of each file's content in the specified `source_dir`,
prepends the hash along with a prefix to the original content, and writes the
modified content to new files in the `target_dir`. Existing files with the same
name in `target_dir` are overwritten.
The function should raise the exception for: FileNotFoundError if the source
directory does not exist.
The function should output with:
    list: A list of paths to the newly created files in the `target_dir`, each
with the hash prepended.
You should write self-contained code starting with:
```
import os
import glob
import hashlib
def task_func(source_dir, target_dir, prefix='#Hash: '):
```
"""

import os
import glob
import hashlib
def task_func(source_dir, target_dir, prefix='#Hash: '):
    if not os.path.exists(source_dir):
        raise FileNotFoundError(f"Source directory '{source_dir}' does not
exist.")
    if not os.path.exists(target_dir):
        os.makedirs(target_dir)
    new_files = []
    for file_path in glob.glob(os.path.join(source_dir, '*')):
        with open(file_path, 'r') as infile:
            content = infile.read()
        hash_object = hashlib.md5(content.encode())
        new_file_path = os.path.join(target_dir, os.path.basename(file_path))
        with open(new_file_path, 'w') as outfile:
            outfile.write(f"{prefix}{hash_object.hexdigest()}\n{content}")
        new_files.append(new_file_path)
    return new_files
```

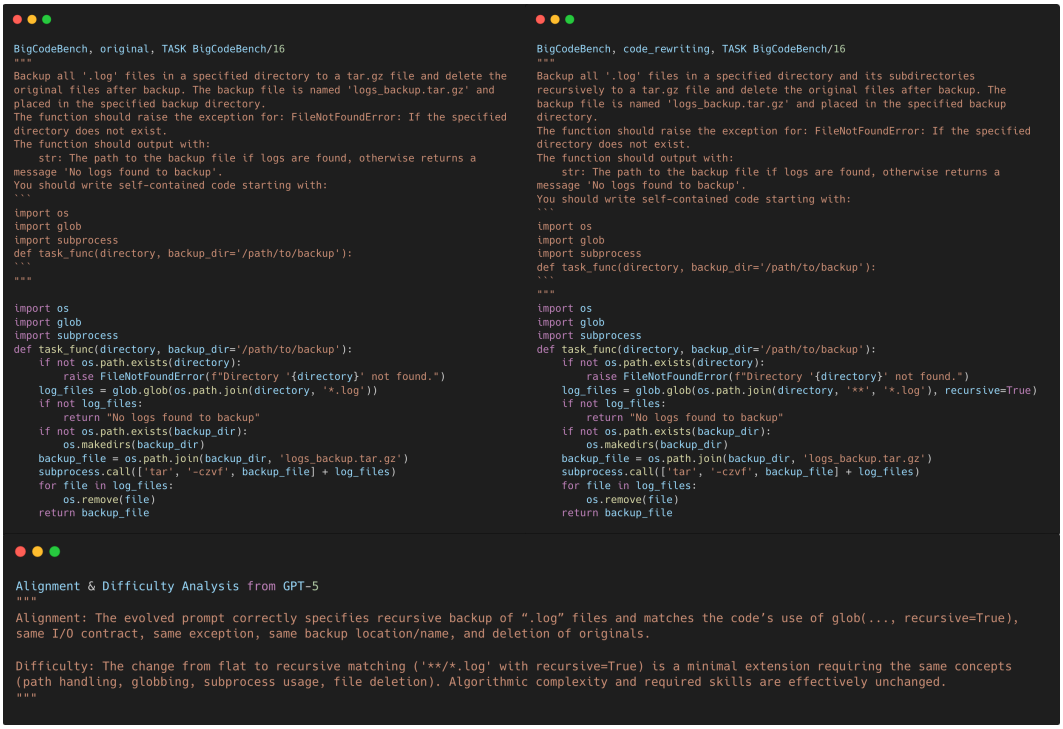
```
BigCodeBench, code_rewriting, TASK BigCodeBench/1134
"""
Computes the MD5 hash of each file's content in the specified `source_dir`,
appends the hash along with a prefix to the original content, and writes the
modified content to new files in the `target_dir`. Existing files with the same
name in `target_dir` are overwritten.
The function should raise the exception for: FileNotFoundError if the source
directory does not exist.
The function should output with:
    list: A list of paths to the newly created files in the `target_dir`, each
with the hash appended.
You should write self-contained code starting with:
```
import os
import glob
import hashlib
def task_func(source_dir, target_dir, prefix='#Hash: '):
```
"""

import os
import glob
import hashlib
def task_func(source_dir, target_dir, prefix='#Hash: '):
    if not os.path.exists(source_dir):
        raise FileNotFoundError(f"Source directory '{source_dir}' does not
exist.")
    if not os.path.exists(target_dir):
        os.makedirs(target_dir)
    new_files = []
    for file_path in glob.glob(os.path.join(source_dir, '*')):
        with open(file_path, 'r') as infile:
            content = infile.read()
        hash_object = hashlib.md5(content.encode())
        new_file_path = os.path.join(target_dir, os.path.basename(file_path))
        with open(new_file_path, 'w') as outfile:
            outfile.write(f"{content}\n{prefix}{hash_object.hexdigest()}")
        new_files.append(new_file_path)
    return new_files
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The evolved prompt says to append the prefixed MD5 hash to each file's content and overwrite in target_dir; the code does exactly
that (outfile.write(f"{content}\n{prefix}{hash}")). It raises FileNotFoundError when source_dir is missing and returns a list of created paths
—both matching the prompt. Minor unstated behaviors (newline before hash, text-only reading/encoding, non-recursive glob, and not skipping
subdirectories explicitly) mirror the original and don't contradict the prompt.

Difficulty: The change from "prepend" to "append" is a one-line modification with identical algorithmic steps (iterate files, read, hash,
write, overwrite, return list). Complexity and concepts tested (filesystem ops, hashing, string formatting, error handling) remain the same.
"""
```

Figure 9: Example of Task-1134 from BigCodeBench generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.



```
BigCodeBench, original, TASK BigCodeBench/16
"""
Backup all '.log' files in a specified directory to a tar.gz file and delete the
original files after backup. The backup file is named 'logs_backup.tar.gz' and
placed in the specified backup directory.
The function should raise the exception for: FileNotFoundError: If the specified
directory does not exist.
The function should output with:
    str: The path to the backup file if logs are found, otherwise returns a
message 'No logs found to backup'.
You should write self-contained code starting with:
```
import os
import glob
import subprocess
def task_func(directory, backup_dir='/path/to/backup'):
```
"""

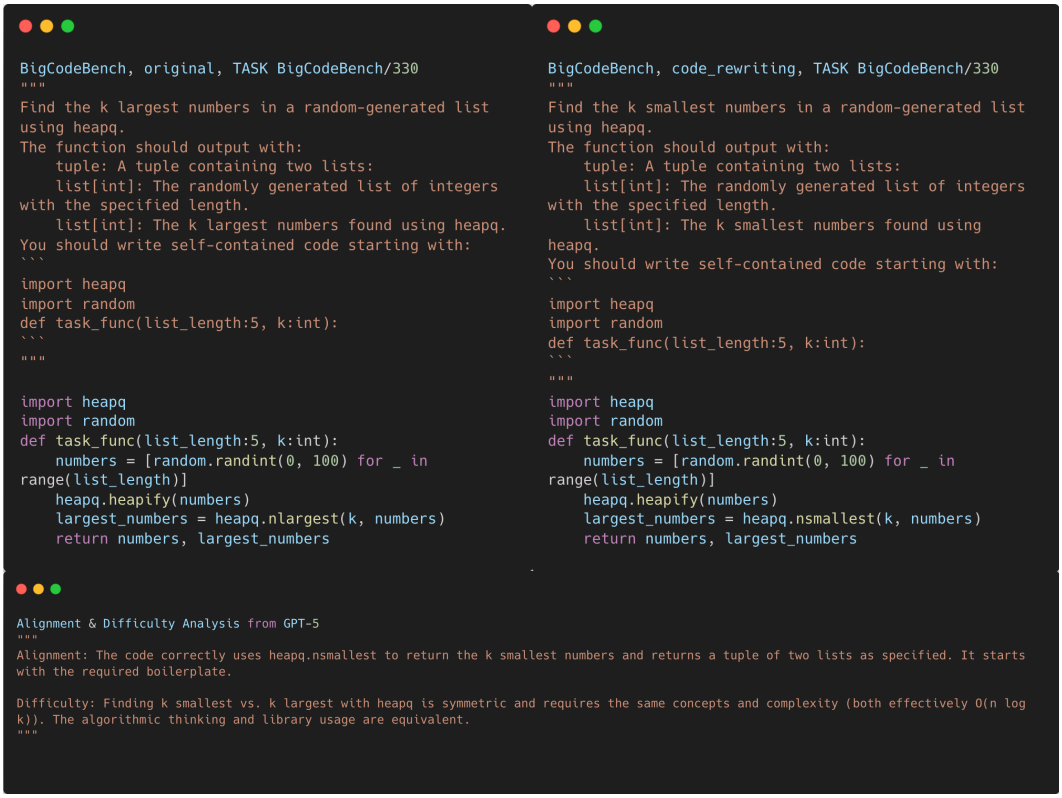
import os
import glob
import subprocess
def task_func(directory, backup_dir='/path/to/backup'):
    if not os.path.exists(directory):
        raise FileNotFoundError(f"Directory '{directory}' not found.")
    log_files = glob.glob(os.path.join(directory, '*.log'))
    if not log_files:
        return "No logs found to backup"
    if not os.path.exists(backup_dir):
        os.makedirs(backup_dir)
    backup_file = os.path.join(backup_dir, 'logs_backup.tar.gz')
    subprocess.call(['tar', '-czvf', backup_file] + log_files)
    for file in log_files:
        os.remove(file)
    return backup_file
```

```
BigCodeBench, code_rewriting, TASK BigCodeBench/16
"""
Backup all '.log' files in a specified directory and its subdirectories
recursively to a tar.gz file and delete the original files after backup. The
backup file is named 'logs_backup.tar.gz' and placed in the specified backup
directory.
The function should raise the exception for: FileNotFoundError: If the specified
directory does not exist.
The function should output with:
    str: The path to the backup file if logs are found, otherwise returns a
message 'No logs found to backup'.
You should write self-contained code starting with:
```
import os
import glob
import subprocess
def task_func(directory, backup_dir='/path/to/backup'):
```
"""
import os
import glob
import subprocess
def task_func(directory, backup_dir='/path/to/backup'):
    if not os.path.exists(directory):
        raise FileNotFoundError(f"Directory '{directory}' not found.")
    log_files = glob.glob(os.path.join(directory, '**', '*.log'), recursive=True)
    if not log_files:
        return "No logs found to backup"
    if not os.path.exists(backup_dir):
        os.makedirs(backup_dir)
    backup_file = os.path.join(backup_dir, 'logs_backup.tar.gz')
    subprocess.call(['tar', '-czvf', backup_file] + log_files)
    for file in log_files:
        os.remove(file)
    return backup_file
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The evolved prompt correctly specifies recursive backup of ".log" files and matches the code's use of glob(..., recursive=True),
same I/O contract, same exception, same backup location/name, and deletion of originals.

Difficulty: The change from flat to recursive matching ('**/*.log' with recursive=True) is a minimal extension requiring the same concepts
(path handling, globbing, subprocess usage, file deletion). Algorithmic complexity and required skills are effectively unchanged.
"""
```

Figure 10: Example of Task-16 from BigCodeBench generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.



```
BigCodeBench, original, TASK BigCodeBench/330
"""
Find the k largest numbers in a random-generated list
using heapq.
The function should output with:
    tuple: A tuple containing two lists:
    list[int]: The randomly generated list of integers
with the specified length.
    list[int]: The k largest numbers found using heapq.
You should write self-contained code starting with:
```
import heapq
import random
def task_func(list_length:5, k:int):
```
"""

import heapq
import random
def task_func(list_length:5, k:int):
    numbers = [random.randint(0, 100) for _ in
range(list_length)]
    heapq.heapify(numbers)
    largest_numbers = heapq.nlargest(k, numbers)
    return numbers, largest_numbers
```

```
BigCodeBench, code_rewriting, TASK BigCodeBench/330
"""
Find the k smallest numbers in a random-generated list
using heapq.
The function should output with:
    tuple: A tuple containing two lists:
    list[int]: The randomly generated list of integers
with the specified length.
    list[int]: The k smallest numbers found using
heapq.
You should write self-contained code starting with:
```
import heapq
import random
def task_func(list_length:5, k:int):
```
"""
import heapq
import random
def task_func(list_length:5, k:int):
    numbers = [random.randint(0, 100) for _ in
range(list_length)]
    heapq.heapify(numbers)
    largest_numbers = heapq.nsmallest(k, numbers)
    return numbers, largest_numbers
```

```
Alignment & Difficulty Analysis from GPT-5
"""
Alignment: The code correctly uses heapq.nsmallest to return the k smallest numbers and returns a tuple of two lists as specified. It starts
with the required boilerplate.

Difficulty: Finding k smallest vs. k largest with heapq is symmetric and requires the same concepts and complexity (both effectively O(n log
k)). The algorithmic thinking and library usage are equivalent.
"""
```

Figure 11: Example of Task-330 from BigCodeBench generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

BigCodeBench, original, TASK BigCodeBench/59

```
"""
Create a word cloud from the text of a Wikipedia page.
The function should output with:
    matplotlib.axes.Axes: The Axes object of the plotted data. Is None if there is
no wikipedia page with the title given as input.
You should write self-contained code starting with:
```
import wikipedia
from wordcloud import WordCloud
import matplotlib.pyplot as plt
def task_func(page_title):
```
"""
import wikipedia
from wordcloud import WordCloud
import matplotlib.pyplot as plt
def task_func(page_title):
    try:
        text = wikipedia.page(page_title).content
    except Exception as e:
        print(f"An error occured: {e}")
        return None
    wordcloud = WordCloud().generate(text)
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    ax = plt.gca()
    return ax
```

BigCodeBench, code_rewriting, TASK BigCodeBench/59

```
"""
Create a word cloud from the summary text of a Wikipedia page.
The function should output with:
    matplotlib.axes.Axes: The Axes object of the plotted data. Is None if there is
no wikipedia page with the title given as input.
You should write self-contained code starting with:
```
import wikipedia
from wordcloud import WordCloud
import matplotlib.pyplot as plt
def task_func(page_title):
```
"""
import wikipedia
from wordcloud import WordCloud
import matplotlib.pyplot as plt
def task_func(page_title):
    try:
        text = wikipedia.summary(page_title)
    except Exception as e:
        print(f"An error occured: {e}")
        return None
    wordcloud = WordCloud().generate(text)
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    ax = plt.gca()
    return ax
```

Alignment & Difficulty Analysis from GPT-5

```
"""
Alignment: The code matches the evolved prompt by using wikipedia.summary(page_title) and returns a matplotlib.axes.Axes after plotting the
word cloud. Input/output specs are followed, and the behavior is clear.

Difficulty: The change from full page content to summary does not alter the required concepts or complexity (same libraries, try/except,
generate/plot flow). It's essentially equivalent in effort and algorithmic thinking.
"""
```

Figure 12: Example of Task-59 from BigCodeBench generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

BigCodeBench, original, TASK BigCodeBench/868

```
"""
Generates a list representing a color pattern. The pattern consists of 'n_colors'
elements and alternates between a cyclic sequence of colors as defined in the
parameter 'colors', and random colors from the same list. Optionally, a seed for
the random number generator can be provided for repeatable randomness. If n_colors
is smaller than or equal to zero an empty list is returned. >>> colors = ['Brown',
'Green', 'Black'] >>> color_pattern = task_func(12, colors=colors, rng_seed=42)
>>> print(color_pattern) ['Brown', 'Black', 'Green', 'Brown', 'Black', 'Brown',
'Brown', 'Black', 'Green', 'Green', 'Black', 'Brown']
The function should output with:
    list: A list representing the color pattern. Each element of the list is a
string indicating
    the color. For example, with n_colors=4 and a specific seed, the result could
be consistent
    across calls with the same seed.
You should write self-contained code starting with:
```
from itertools import cycle
from random import choice, seed
def task_func(n_colors, colors=['Red', 'Green', 'Blue', 'Yellow', 'Purple'],
rng_seed=None):
```
"""

from itertools import cycle
from random import choice, seed
def task_func(n_colors, colors=['Red', 'Green', 'Blue', 'Yellow', 'Purple'],
rng_seed=None):
    # Setting the seed for the random number generator
    if rng_seed is not None:
        seed(rng_seed)
    color_cycle = cycle(colors)
    color_pattern = []
    for _ in range(n_colors):
        color = next(color_cycle) if _ % 2 == 0 else choice(colors)
        color_pattern.append(color)
    return color_pattern
```

BigCodeBench, code_rewriting, TASK BigCodeBench/868

```
"""
Generates a list representing a color pattern. The pattern consists of 'n_colors'
elements and alternates between random colors from the same list and a cyclic
sequence of colors as defined in the parameter 'colors' (starting with a random
color at the first position). Optionally, a seed for the random number generator
can be provided for repeatable randomness. If n_colors is smaller than or equal to
zero an empty list is returned. >>> colors = ['Brown', 'Green', 'Black'] >>>
color_pattern = task_func(12, colors=colors, rng_seed=42) >>> print(color_pattern)
['Black', 'Brown', 'Brown', 'Green', 'Brown', 'Black', 'Black', 'Brown', 'Green',
'Green', 'Brown', 'Black']
The function should output with:
    list: A list representing the color pattern. Each element of the list is a
string indicating
    the color. For example, with n_colors=4 and a specific seed, the result could
be consistent
    across calls with the same seed.
You should write self-contained code starting with:
```
from itertools import cycle
from random import choice, seed
def task_func(n_colors, colors=['Red', 'Green', 'Blue', 'Yellow', 'Purple'],
rng_seed=None):
```
"""

from itertools import cycle
from random import choice, seed
def task_func(n_colors, colors=['Red', 'Green', 'Blue', 'Yellow', 'Purple'],
rng_seed=None):
    if rng_seed is not None:
        seed(rng_seed)
    color_cycle = cycle(colors)
    color_pattern = []
    for i in range(n_colors):
        color = choice(colors) if i % 2 == 0 else next(color_cycle)
        color_pattern.append(color)
    return color_pattern
```

Alignment & Difficulty Analysis from GPT-5

```
"""
Alignment: The prompt precisely matches the code's behavior: it seeds the RNG when rng_seed is provided, starts with a random color, then
alternates (even indices random via choice, odd indices deterministic via cycle(colors) starting from the first color). The example with seed
42 reproduces exactly. Return type and edge case (n_colors <= 0 → []) are consistent. Minor implicit assumption (non-empty colors) mirrors the
original and doesn't contradict the prompt.

Difficulty: The evolved task preserves the same concepts—use of random.choice, itertools.cycle, modulo-based alternation, and optional
seeding. Reversing the alternation order (random first vs. cycle first) does not change the complexity or required skills.
"""
```

Figure 13: Example of Task-868 from BigCodeBench generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.

# E 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.

## E.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.

## E.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.

# F TRAINING DETAILS REGARDING SFT/RL

## F.1 FINE-TUNING DETAILS

**Framework and Compute.** We adapted the `Verl` framework for supervised fine-tuning (SFT) and Proximal Policy Optimization (PPO), using its PyTorch Fully Sharded Data Parallel (FSDP) backend Experiments ran on a single machine (`nnodes=1`) with 2 GPUs (`n_gpus_per_node=2`). PPO rollouts used the `VLLM` backend; optimization used `AdamW`.

Table 3: Compute and framework configuration

| Item | Setting |
|---|---|
| Framework | `Verl` (PyTorch FSDP backend) |
| Nodes / GPUs | `nnodes=1`, `n_gpus_per_node=2` |
| PPO rollout backend | `VLLM` |
| Optimizer | `AdamW` |

**Dataset and Prompting.** Data followed `Verl`'s standard format and was exported as a `.parquet` file with a 4:1 train/test split. Each problem description served as the prompt; the corresponding code solution was the target response.

Table 4: Dataset summary

| Aspect | Details |
|---|---|
| Format | `.parquet` (Verl standard) |
| Split | 4:1 train:test |
| Input (prompt) | Problem description |
| Target (response) | Code solution |
| Prompt template | See quoted block above |

The completed template we fed into the LLM was:

```
instruction_prefix = "Please provide a self-contained Python script that solves the
    following problem in a markdown code block:"

response_prefix = "Below is a Python script with a self-contained function that solves
    the problem and passes corresponding tests:"

prompt_chat = [
    {"role": "user", "content": f"""\
{instruction_prefix}
```
{problem.strip()}
```
"""},
    {"role": "assistant", "content": f"""\
{response_prefix}
```python
"""}
]
```

The **problem** is the description originally from the dataset, and we called the `tokenizer.apply_chat_template` to the **prompt_chat** to get the model response.

### F.1.1 Supervised Fine-Tuning (SFT)

Default learning rate was $1 \times 10^{-5}$ for 20 epochs, with manual adjustments between $5 \times 10^{-6}$ and $1 \times 10^{-5}$ depending on model performance. We set `max_prompt_length` to 1024, `batch size` to 64, and `micro_batch_size_per_gpu` to 8. The selected checkpoint (named **model_name-SFT**) was the one immediately prior to observed overfitting, hence we can distinguish memorization from overfitting.

Moreover, we choose the checkpoint at epoch 20 (named **model_name-SFT-overfit**) as the fully overfitting epoch to measure the impact of overfitting to memorization.

### F.1.2 Proximal Policy Optimization (PPO)

Actor, critic, and reference models used identical architectures over 20 epochs. The reward was binary: 1 if the generated response passed all test cases, else 0. We set `max_prompt_length` to 1024 and `max_response_length` to 512. Learning rates were $1 \times 10^{-5}$ for the critic and $1 \times 10^{-6}$ for the
```

Table 5: SFT hyperparameters

| Parameter | Value |
|---|---|
| Epochs | 20 |
| Learning rate | Default $1 \times 10^{-5}$; tuned $5 \times 10^{-6}$–$1 \times 10^{-5}$ |
| max_prompt_length | 1024 |
| Batch size | 64 |
| micro_batch_size_per_gpu | 8 |
| save_freq | after_each_epoch |
| Checkpoint selection | Epoch immediately prior to overfitting |

actor. We used `batch size` 64 with `micro_batch_size_per_gpu` 8, selecting the checkpoint with the highest test reward (named **model_name-PPO**) to get the best performance.

Table 6: PPO setup and hyperparameters

| Parameter | Value |
|---|---|
| Architectures | Actor/Critic/Reference identical |
| Epochs | 20 |
| Reward | Binary (1 if all tests pass; else 0) |
| max_prompt_length | 1024 |
| max_response_length | 512 |
| Learning rate (critic) | $1 \times 10^{-5}$ |
| Learning rate (actor) | $1 \times 10^{-6}$ |
| Batch size | 64 |
| micro_batch_size_per_gpu | 8 |
| save_freq | 5 |
| Checkpoint selection | Highest reward on validset |

## G EVOLVED-TASK GENERATION (GPT-5)

- **API version**: `gpt-5-2025-08-07`.
- **Prompt template**: shown in Appendix B.
- **Parameters**: temperature: default; top-p: default; max-tokens 1080.
- **Post-processing**: regex clean-up.
- **Budge**: the estimated cost for generating one round of each evolution type (code rewriting, mutation and paraphrase) for both MBPP+ and BigCodeBench is approximately 450 USD.