Evaluating Harmful Memorization in LLM Code Generation

Anonymous ACL submission

Abstract

Assessing memorization in LLM code generation is challenging because code reuse is not inherently harmful. Correct programs often share APIs, templates, and algorithmic idioms, so code similarity alone may reflect benign reuse rather than problematic memorization. We focus on the harmful case: a model fails to adapt when task semantics change and instead produces code that remains highly similar to the original solution. We propose *code rewriting*, an automated semantic perturbation pipeline that constructs superficially similar but semantically different variants of existing coding tasks by modifying a core logic element of the reference solution and regenerating the corresponding prompt. To summarize this behavior, we introduce the *Harmful Memorization Index* (HMI), an operational diagnostic that combines similarity to the original solution with performance degradation on rewritten tasks. HMI is intended to complement standard accuracy metrics by capturing cases where benchmark success is brittle under small semantic changes. Across MBPP+ and BigCodeBench, we find that: (1) harmful memorization does not monotonically increase with model scale and often decreases on simpler tasks; (2) failures are concentrated in specific logic-edit types, especially constraint changes and data transformations; (3) supervised fine-tuning improves original-task accuracy but can substantially amplify harmful memorization; and (4) PPO yields a more favorable accuracy–HMI trade-off, mitigating harmful memorization while maintaining competitive performance.

1 Introduction

Large language models (LLMs) have made rapid progress in code generation and are increasingly used as practical tools in software development (Sourcegraph, 2024; Tabnine, 2024; Team et al., 2023; Anthropic, 2025; Chen et al., 2021). Modern code-focused LLMs achieve strong performance on programming benchmarks (Rozière et al., 2024; Hui et al., 2024), making benchmark accuracy an essential measure of code-generation capability. However, accuracy alone does not reveal how a model behaves when a familiar-looking programming task requires a different solution.

This distinction is important because memorization in code generation is not uniformly harmful. Many correct programs reuse common APIs, boilerplate, library conventions, programming idioms, or canonical algorithmic templates. Penalizing code similarity alone would therefore mischaracterize useful knowledge reuse as undesirable memorization. The harmful case is more specific: a model may continue to rely on a familiar solution pattern even after the task semantics have changed, producing code that resembles the original solution but no longer satisfies the new requirement. In this setting, the key question is not whether the generated code is similar to existing code, but whether such similarity remains semantically appropriate.

We study this behavior as *harmful memorization*: a model succeeds on an original task, but fails to adapt to a semantically rewritten variant while producing code that remains highly similar to the original solution. To operationalize this behavior, we introduce *code rewriting*, an automated semantic perturbation pipeline that constructs near-neighbor coding tasks. Given an original task and its reference solution, the pipeline modifies one core logic element of the solution while preserving much of its surrounding structure, and then regenerates a corresponding task prompt. The resulting rewritten task is superficially similar to the original task but requires different underlying logic. This controlled setup allows us to ask: when the surface form remains familiar but the semantics change, does the model adapt its solution, or does it regress to the original solution pattern?

We summarize this behavior with the *Harmful Memorization Index* (HMI), an operational diag-

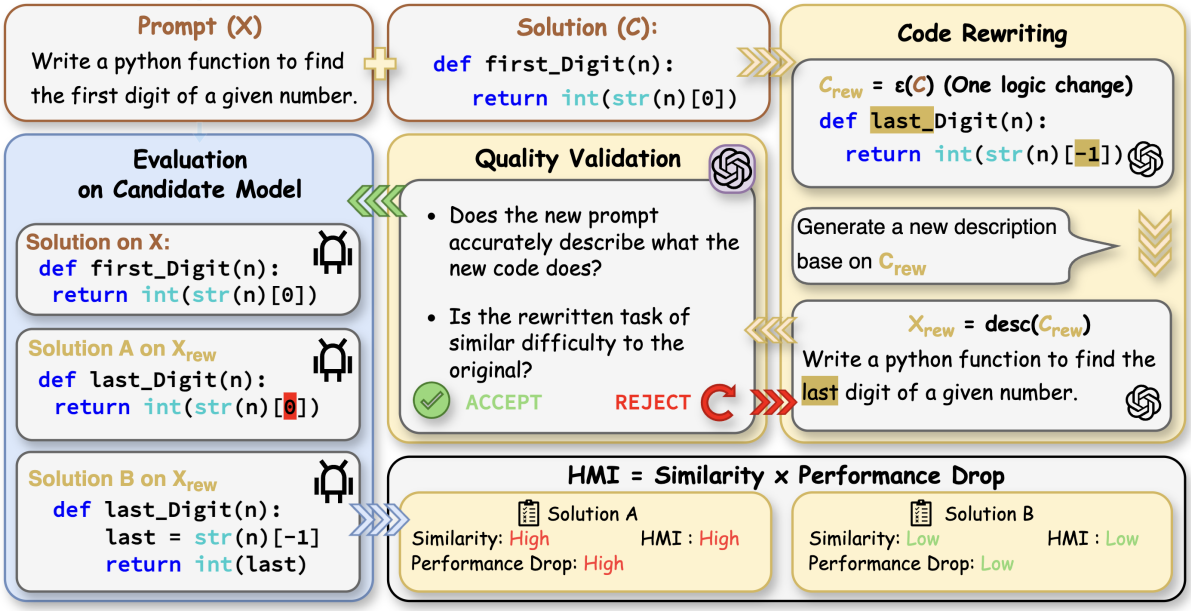

Figure 1: Overview of the automated code rewriting and harmful memorization evaluation pipeline. Code rewriting first changes a single logic element within the ground-truth solution C while preserving the original function's structure. It then generates a new prompt X_{rew} based on C_{rew}. An LLM-as-a-judge verifies that C_{rew} matches X_{rew} and aligns with the original task's difficulty. Finally, the candidate model's output is evaluated using the Harmful Memorization Index (HMI), which combines the structural and semantic similarity of the model's answer with its relative performance drop on the rewritten task.

nostic that combines two signals: (i) the similarity between the model's answer on the rewritten task and the original reference solution, and (ii) the relative performance drop from the original task to the rewritten task. HMI is most informative when interpreted alongside standard accuracy metrics. A model with high original-task accuracy and low HMI both solves the benchmark and adapts to semantic rewrites. In contrast, a model with high original-task accuracy and high HMI may perform well on the original benchmark while remaining brittle under small but consequential semantic changes. Low HMI should not by itself be interpreted as strong coding ability, since a model with low original-task accuracy may simply fail without reproducing the original solution pattern.

Our evaluation focuses on function-level coding benchmarks across different difficulty levels, from simpler problems in MBPP+ (Liu et al., 2023) to more difficult tasks in BigCodeBench (Zhuo et al., 2024). This scope enables controlled semantic rewrites and fine-grained attribution of failures to localized logic changes, such as constraint changes, data transformations, component swaps, and logic reversals. Using this setup, we evaluate multiple model families and post-training strategies. Fine-tuning is not required to compute HMI: we first apply the diagnostic directly to existing instruction-tuned and code-focused models. We then use Supervised Fine-Tuning (SFT) and Proximal Policy Optimization (PPO) as additional case studies to analyze how post-training changes the accuracy–HMI trade-off. Our study shows that HMI does not monotonically increase with scale, is concentrated in specific logic-edit types such as constraint changes and data transformations, and changes substantially under post-training, with SFT increasing HMI more than PPO.

In summary, our work makes the following contributions:

- We formulate harmful memorization in code generation behaviorally as: failure on a semantically rewritten task under high similarity to the original solution, rather than similarity alone.
- We propose an automated **code-rewriting diagnostic** that constructs superficially similar but semantically different variants of existing coding tasks by modifying localized solution logic.
- We introduce HMI as an operational index for analyzing harmful memorization alongside standard accuracy metrics.
- We conduct an empirical study across model scale, benchmark difficulty, logic-change types, and post-training strategies, showing when code

models adapt to changed semantics and when they regress to familiar solution patterns.

2 Related Work

LLMs for Code Generation. Recent LLMs achieve strong results on code-generation benchmarks such as BIGCODEBENCH, LiveCodeBench, and EvalPlus (Zhuo et al., 2024; Jain et al., 2024; Liu et al., 2023). These benchmarks provide increasingly rigorous tests of functional correctness, task diversity, and contamination-aware evaluation. Our work is complementary: rather than replacing functional correctness evaluation, we use semantic rewriting to analyze how models behave when tasks remain familiar but require different solutions.

Memorization and Code Reuse. Prior work studies memorization in language and code models through extraction attacks, prefix-to-suffix completion, clone detection, data contamination analysis, and overlap with training corpora (Carlini et al., 2019; Pappu et al., 2024; Duan et al., 2024; Kassem et al., 2024; Hartmann et al., 2023; Riddell et al., 2024; Dong et al., 2024; Xie et al., 2024; Lu et al., 2024; Staab et al., 2023; Zanella-Béguelin et al., 2020; Chen et al., 2021). These studies show that models can reproduce or closely resemble seen content, and several works report that measured memorization can increase with model size (Yang et al., 2024; Al-Kaswan et al., 2024; Wang et al., 2024). However, in code generation, similarity is difficult to interpret: correct programs often share APIs, boilerplate, library conventions, and canonical algorithmic structures. As a result, similarity-based memorization measures can conflate harmful rote reuse with benign code reuse that is necessary for correct programming.

Recent work has therefore emphasized that memorization should be interpreted through its effect on downstream behavior rather than through overlap alone (Bayat et al., 2024). Closest to our setting, Lai et al. (2022) use semantic perturbations to probe whether models adapt when reference-solution semantics change without increasing task difficulty. Our work follows this behavioral view but focuses on general-purpose function-level code-generation benchmarks. We introduce an automated *code-rewriting* diagnostic that constructs superficially similar but semantically different task variants, and we summarize model behavior with HMI, an operational index combining similarity to the original solution with performance degradation on the rewritten task. This allows us to analyze harmful memorization as failure under high similarity, while separating it from benign reuse where similar code remains correct.

3 Methodology

3.1 Code Rewriting

To probe harmful memorization, we use *code rewriting* to generate problems that are **semantically different** from the original tasks while remaining superficially similar. This automated pipeline is shown in Figure 1. Specifically, we first modify **one core logic element** in the reference solution while preserving the original function structure. We then generate a new task description that reflects the altered code while remaining syntactically similar to the original task. Formally, let $x \in X$ be the original prompt in text space X and $c \in C$ its reference code solution in code space C. We apply a rewriting function ϵ that produces a rewritten solution $c_{\text{rew}} = \epsilon(c)$, where c_{rew} is functionally different from c but shares the same function signature. The new prompt x_{rew} is then generated from c_{rew}:

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

$$\text{sig}(c_{\text{rew}}) = \text{sig}(c), \quad c_{\text{rew}} \not\equiv c, \tag{2}$$

where $\text{desc}(\cdot)$ generates a task description from code and $\text{sig}(\cdot)$ extracts the function signature.

Data Validation. We validate rewritten tasks with three checks. First, GPT-5 (OpenAI, 2025) verifies prompt–solution consistency and comparable difficulty; this LLM judge is used only for dataset validation, not for scoring candidate models. Second, each rewritten solution is executed against its corresponding rewritten unit tests (see Appendix B). Third, four experienced Python programmers manually reviewed generated examples for prompt–solution consistency and semantic plausibility. Representative regressed examples are shown in Appendix G.

3.1.1 Harmful Memorization Index

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

(i) Similarity For every rewritten task $i \in \mathcal{T}_{\text{rew}}$, we measure how closely the model-generated solution for the rewritten prompt resembles the original reference solution. We use two complementary, deterministic similarity measures:

- **AST similarity**: normalized tree-edit overlap between abstract syntax trees, capturing structural resemblance.
- **Edit similarity**: $1 - \frac{\text{Levenshtein distance}}{\text{max length}}$ over tokenized code, capturing surface-form resemblance.

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

$$S_i = \frac{\text{AST}_i + \text{Edit}_i}{2}. \quad (3)$$

At the corpus level, we define the mean similarity over all rewritten tasks as:

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

(ii) Relative Accuracy Drop 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

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

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

Corpus-Level HMI and Instance-Level iHMI We formulate the final index in two complementary ways. Our primary metric, the corpus-level HMI, is defined as the product of the mean similarity and the overall relative accuracy drop:

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

Unlike an additive design, this multiplicative design makes HMI high only when both diagnostic conditions hold: (i) the model output for the rewritten task remains highly similar to the original reference solution and (ii) performance drops on the rewritten tasks. Thus, high HMI indicates failure under high similarity, while low HMI should be interpreted together with original-task accuracy.

While corpus-level HMI is convenient for comparing models, it combines corpus-level averages. We therefore also report an instance-level variant, iHMI, which directly couples similarity and failure on the same examples:

$$\text{iHMI} = \frac{1}{|\mathcal{T}_{\text{passed}}|} \sum_{k \in \mathcal{T}_{\text{passed}}} \left(S_k \cdot \text{Drop}_k\right), \quad (7)$$

where $\mathcal{T}_{\text{passed}}$ is the set of tasks that the model solves correctly on the original benchmark, S_k is the unified similarity score for example k, and $\text{Drop}_k \in \{0, 1\}$ indicates whether the model fails on the rewritten task. Conditioning on $\mathcal{T}_{\text{passed}}$ reflects on regression under semantic rewriting: failure on a rewritten task is most informative when the model solved the corresponding original task.

Accounting for low baseline accuracy. Because RAD_{rew} is normalized by original-task accuracy, it can be unstable when $\text{Acc}(\mathcal{T}_{\text{ori}})$ is very low. We therefore treat HMI as meaningful only for model–benchmark pairs with sufficient original-task competence. In our main comparisons, we report RAD_{rew} and HMI when $\text{Acc}(\mathcal{T}_{\text{ori}}) \geq 10\%$ and the model solves at least 50 original tasks. For lower-competence settings, we still report $\text{Acc}(\mathcal{T}_{\text{ori}})$ and $\text{Acc}(\mathcal{T}_{\text{rew}})$, but mark RAD_{rew} and HMI as "N/A" to avoid over-interpreting noisy relative drops. We provide additional justification in Appendix E.

3.2 Post-Training Case Studies

Supervised Fine-Tuning In SFT, coding problems are used as inputs and reference solutions as targets. We train models on the training split and evaluate HMI on the held-out test split. Since prolonged SFT can improve training-set fit while potentially reducing robustness to semantic rewrites, we evaluate multiple checkpoints around the validation-loss optimum. This allows us to examine whether additional accuracy gains are accompanied by higher HMI.

Reinforcement Learning We also consider PPO (Schulman et al., 2017) as an alternative post-training strategy. PPO optimizes a reward based on functional correctness of generated code while regularizing updates through a reference policy. We use the same model architecture for the actor, critic, and reference models, and select the checkpoint with the highest validation reward for evaluation. Comparing PPO with SFT allows us to analyze whether post-training methods that optimize executable correctness induce different accuracy–HMI trade-offs.

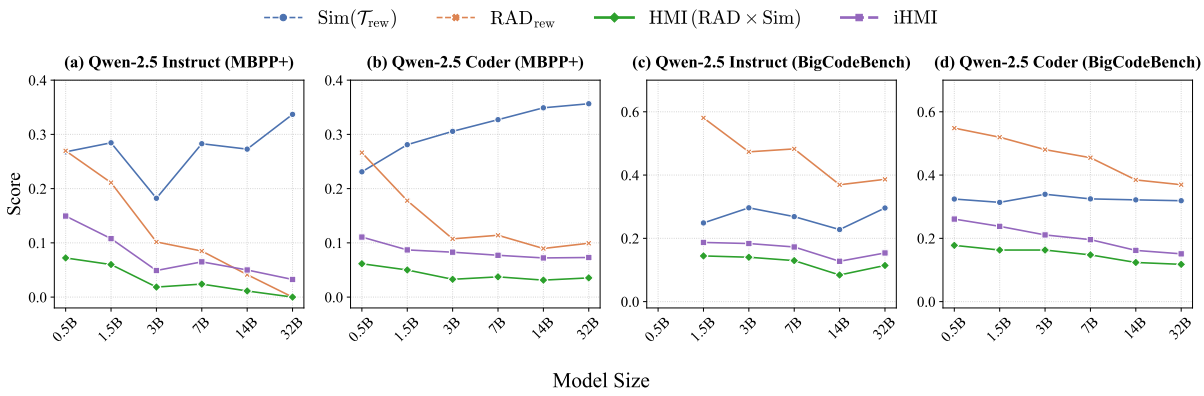

Figure 2: Scaling trends in HMI across Qwen-2.5 family. The Qwen-2.5 Instruct-0.5B on BIGCODEBENCH is omitted because its baseline Pass@1 falls below our competence threshold (Appendix E).

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 a 4:1 train/test split for fine-tuning. For models without fine-tuning, we use the **complete set** of tasks for evaluation. For SFT and PPO, we train on the **training split** and evaluate on the **test split**[1]. Due to the small size of MBPP+ test split ($n = 78$), estimates on this split may be noisy, we only use BIGCODEBENCH to explore the impact of fine-tuning strategies on HMI.

Code Rewriting We generate one perturbed variant for each original task. More about the generation process is given in Appendix D.

4.2 Models

In this paper, we conduct our 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 four NVIDIA A100 GPUs (80GB), with a total computational budget of approximately 40 GPU hours, using PyTorch and HuggingFace Transformers. More training details can be found at Appendix C.

[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.

5 Results

5.1 Main results

Logic Change Type	BigCodeBench	MBPP+
Component Swap	393 (34.5%)	52 (13.8%)
Data Transformation	385 (33.8%)	58 (15.3%)
Constraint Change	185 (16.2%)	86 (22.8%)
Logic Reversal	50 (4.4%)	68 (18.0%)
Workflow Modification	50 (4.4%)	10 (2.6%)
Constant Swap	37 (3.3%)	36 (9.5%)
Operation Order Swap	20 (1.8%)	12 (3.2%)
Operator Change	18 (1.6%)	36 (9.5%)
Other	2 (0.2%)	20 (5.3%)
Total	**1140 (100.0%)**	**378 (100.0%)**

Table 1: Statistics of Code Rewriting logic change types for BigCodeBench and MBPP+.

Harmful memorization does not monotonically increase with model scale. Figure 2 shows the results for the Qwen-2.5 family on HMI and iHMI. Across both Qwen-2.5 Instruct and Qwen-2.5 Coder Instruct models, scaling is generally associated with lower RAD_{rew} and hence lower HMI. This suggests that larger models are often better able to adapt to semantic rewrites, even when their outputs remain similar to the original reference solutions. The gains are most pronounced on simpler tasks (MBPP+) and are less consistent on harder tasks (BIGCODEBENCH); on BIGCODEBENCH, non-zero HMI and iHMI are primarily explained by persistently high RAD_{rew} together with relatively stable $\text{Sim}(\mathcal{T}_{\text{rew}})$. Notably, as shown by the $\text{Sim}(\mathcal{T}_{\text{rew}})$ curves in Figure 2(a) and (b), similarity to the original reference solution often remains high or even increases with model size. A similarity-only analysis would therefore suggest that larger models exhibit more memorization. HMI gives a

Model	HMI on MBPP+				
	Component Swap	Constraint Change	Data Transformation	Logic Reversal	**All**
Qwen-2.5-0.5B-Instruct	0.0212	0.0929	0.0877	0.1119	0.0722
Qwen-2.5-1.5B-Instruct	0.0644	0.0584	0.0592	0.0553	0.0600
Qwen-2.5-3B-Instruct	0.0149	0.0380	0.0224	0.0000	0.0185
Qwen-2.5-7B-Instruct	0.0154	0.0543	0.0284	0.0072	0.0240
Qwen-2.5-14B-Instruct	0.0139	0.0573	0.0172	0.0000	0.0113
Qwen-2.5-32B-Instruct	0.0000	0.0000	0.0000	0.0073	0.0000
Mean ± Std	*0.02 ± 0.02*	*0.05 ± 0.03*	*0.04 ± 0.03*	*0.03 ± 0.05*	—
Qwen-2.5-coder-0.5B-Instruct	0.0195	0.0738	0.0480	0.0483	0.0615
Qwen-2.5-coder-1.5B-Instruct	0.0451	0.0465	0.0752	0.0284	0.0500
Qwen-2.5-coder-3B-Instruct	0.0000	0.0380	0.0406	0.0079	0.0328
Qwen-2.5-coder-7B-Instruct	0.0000	0.0382	0.0622	0.0189	0.0373
Qwen-2.5-coder-14B-Instruct	0.0189	0.0828	0.0372	0.0000	0.0313
Qwen-2.5-coder-32B-Instruct	0.0000	0.0786	0.0433	0.0356	0.0354
Mean ± Std	*0.01 ± 0.02*	*0.06 ± 0.02*	*0.05 ± 0.01*	*0.02 ± 0.02*	—

Table 2: HMI across logic change types for Qwen-2.5 Instruct and Coder variants on MBPP+. HMI highlighted in red denotes the highest HMI per row.

Model	HMI on BigCodeBench					
	Component Swap	Constraint Change	Data Transformation	Logic Reversal	Workflow Modification	**All**
Qwen-2.5-0.5B-Instruct	0.1635	0.1968	0.1951	0.2144	0.1900	0.1740
Qwen-2.5-1.5B-Instruct	0.1316	0.1551	0.1539	0.1698	0.0749	0.1444
Qwen-2.5-3B-Instruct	0.1334	0.1442	0.1399	0.1698	0.1558	0.1403
Qwen-2.5-7B-Instruct	0.1178	0.1355	0.1368	0.2118	0.1035	0.1296
Qwen-2.5-14B-Instruct	0.0685	0.0775	0.0998	0.0932	0.0850	0.0841
Qwen-2.5-32B-Instruct	0.0960	0.1130	0.1217	0.1268	0.1335	0.1143
Mean ± Std	*0.12 ± 0.03*	*0.14 ± 0.04*	*0.14 ± 0.03*	*0.16 ± 0.05*	*0.12 ± 0.04*	—
Qwen-2.5-coder-0.5B-Instruct	0.1628	0.1302	0.2060	0.0787	0.1203	0.1778
Qwen-2.5-coder-1.5B-Instruct	0.1302	0.2207	0.1650	0.2135	0.1742	0.1630
Qwen-2.5-coder-3B-Instruct	0.1385	0.1813	0.1848	0.2111	0.0841	0.1630
Qwen-2.5-coder-7B-Instruct	0.1321	0.1440	0.1581	0.1947	0.1590	0.1477
Qwen-2.5-coder-14B-Instruct	0.1192	0.1110	0.1336	0.0956	0.1353	0.1237
Qwen-2.5-coder-32B-Instruct	0.1029	0.1242	0.1276	0.1191	0.1279	0.1178
Mean ± Std	*0.13 ± 0.02*	*0.15 ± 0.04*	*0.16 ± 0.03*	*0.15 ± 0.06*	*0.13 ± 0.03*	—

Table 3: HMI across logic change types for Qwen-2.5 Instruct and coder variants on BigCodeBench. HMI highlighted in red denotes the highest HMI per row.

different view: when high similarity is not accompanied by a large performance drop, it is less indicative of harmful memorization and more consistent with benign reuse of common code structure.

Harmful memorization declines rapidly on simpler tasks but persists on more difficult ones. On the introductory-level tasks in MBPP+, HMI decreases notably as models scale up. For instance, Qwen-2.5-Instruct's HMI falls from Qwen-2.5-0.5B-Instruct to effectively zero at Qwen-2.5-32B-Instruct. Conversely, on the more challenging BIGCODEBENCH, HMI values remain significant at Qwen-2.5-32B-Instruct. This discrepancy shows that while larger models better capture underlying semantics changes, they do not completely eliminate memorization, especially on challenging tasks that demand deeper reasoning.

Coder models increase solution similarity without substantially increasing HMI. Coder models yield higher $\mathrm{Sim}(\mathcal{T}_{\mathrm{rew}})$ than their instruction-only counterparts. For instance, on BIGCODEBENCH, the Qwen-2.5-Coder series scores $\mathbf{0.3237} \pm 0.0086$ compared with $\mathbf{0.2670} \pm 0.0268$ for the instruction-only variant (mean ± SD over 6 seeds). However, $\mathrm{RAD}_{\mathrm{rew}}$ remains comparable across these variants, translating to only a slight increase in HMI ($\mathbf{0.1367} \pm 0.0224$ vs. $\mathbf{0.1142} \pm 0.0247$). This pattern suggests that code-focused pretraining encourages reuse of familiar code structures, but this increased similarity does not necessarily translate into substantially higher harmful memorization.

Model	MBPP+				BigCodeBench			
	$\text{Acc}(\mathcal{T}_{\text{ori}})$ (↑)	$\text{Sim}(\mathcal{T}_{\text{rew}})$ (↓)	RAD_{rew} (↓)	**HMI** (↓)	$\text{Acc}(\mathcal{T}_{\text{ori}})$ (↑)	$\text{Sim}(\mathcal{T}_{\text{rew}})$ (↓)	RAD_{rew} (↓)	**HMI** (↓)
Llama-3.1-8B-Instruct	0.5529	0.1486	0.0133	0.0020	0.3079	0.2132	0.4444	0.0947
Llama-3.1-70B-Instruct	0.6984	0.1518	0.0000	0.0000	0.4175	0.2404	0.3676	0.0884
Llama-4-Scout-17B-Instruct (16E)	0.6614	0.1446	0.0160	0.0023	0.4061	0.2343	0.3909	0.0916
Llama-4-Maverick-17B-Instruct (128E)	0.7751	0.2669	0.0307	0.0082	0.4860	0.2357	0.3953	0.0932

Table 4: HMI for Llama-3.1 and Llama-4 instruct series. HMI persists in harder tasks (BIGCODEBENCH), as RAD_{rew} stays high even as $\text{Sim}(\mathcal{T}_{\text{rew}})$ is comparable.

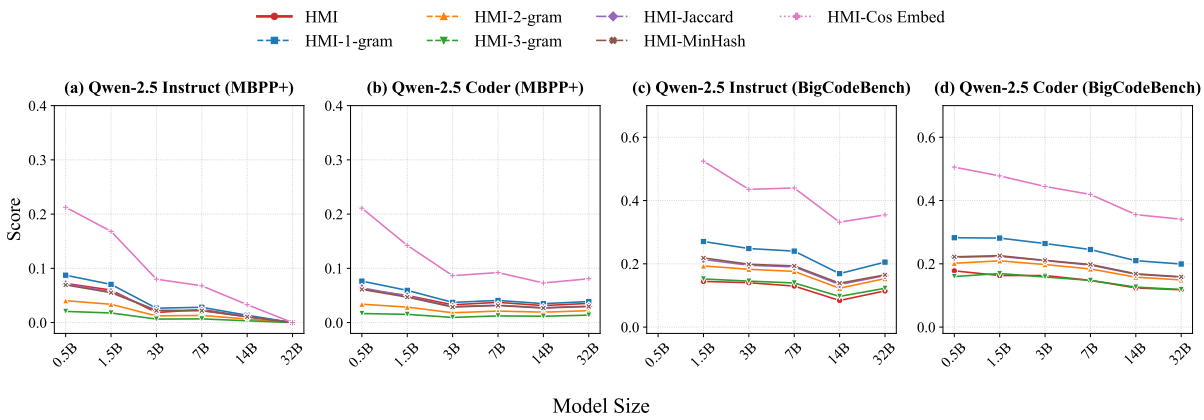

Figure 3: Scaling trends in HMI across Qwen-2.5 with different implementation choices.

5.2 Harmful memorization concentrates in specific logic-edit types.

To better understand which semantic changes are associated with harmful memorization, we annotate each rewritten pair with a fine-grained logic-change category, shown in Table 1. To avoid unstable estimates, we exclude logic-change categories with fewer than 50 samples in a given dataset. Detailed definitions can be found in Appendix F. Table 2 and Table 3 show HMI by logic-change type. On MBPP+ (Table 2), **Constraint Change** and **Data Transformation** consistently exhibit the highest per-type HMI across both Instruct and Coder series, whereas **Component Swap** has much smaller HMI and often approaches zero for larger models. On BIGCODEBENCH (Table 3), all five major types show substantial and relatively flat HMI across scales, with **Data Transformation**, **Logic Reversal**, and **Workflow Modification** frequently attaining the highest per-type HMI.

5.3 Ablation Study

We perform ablation studies to test whether our findings are robust across model families and alternative implementations of the similarity component in HMI.

Across Llama families. We evaluate Llama models in Table 4. The Llama-3.1 models show lower HMI as scale increases, consistent with the Qwen-family trend. The Llama 4 variants[2] show comparable HMI on both datasets. These results suggest that the main scaling pattern is not specific to the Qwen family, although the magnitude of HMI varies across model families and architectures.

Varying similarity functions. Figure 3 evaluates alternative similarity functions, including 1–3-gram overlap, Jaccard similarity, MinHash, and cosine similarity over code embeddings on both MBPP+ and BIGCODEBENCH. HMI trends are stable across these alternatives: Jaccard, MinHash, and 1–3-gram variants closely track our original HMI, while cosine-embedding similarity differs more in absolute scale but preserves the relative model trends. This indicates that our main conclusions are not an artifact of a single similarity implementation.

Varying the weighting between AST-based and edit-based similarity. Figure 4 evaluates different weightings between AST and edit-distance-

[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.

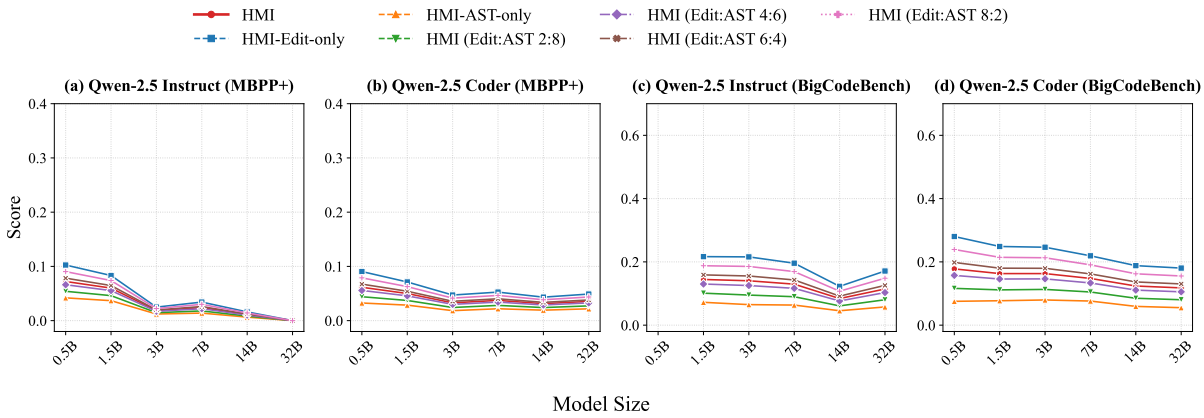

Figure 4: Scaling trends in HMI across Qwen-2.5 family with different weighting between AST-based and edit-based similarity.

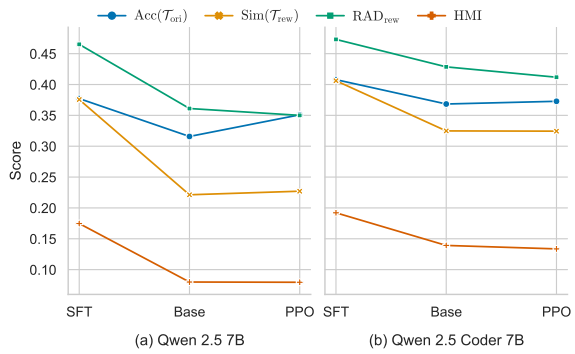

Figure 5: Results from fine-tuning models on BIGCODEBENCH. Checkpoints selected for SFT and PPO follows rules in Section 3.2.

Model	Stage (loss)	Acc($\mathcal{T}_{\text{ori}}$) (↑)	HMI
Qwen-2.5-7B	Early (0.324)	35.96%	0.1540
	Best-Val (0.302)	37.72%	0.1733
	Late (0.326)	39.47%	0.1795
Coder-7B	Early (0.325)	35.96%	0.1537
	Best-Val (0.292)	40.79%	0.1864
	Late (0.318)	42.54%	0.2216

Table 5: Changes in original-task accuracy and HMI during SFT training on BIGCODEBENCH. Accuracy increases steadily from Early to Best-Val to Late, but HMI grows more sharply

based similarity. Across these settings, HMI under our default 50/50 weighting is highly correlated with HMI under alternative weightings. This suggests that our findings are not sensitive to the weighting of structural and edit-based similarity.

5.4 Impact of Fine-Tuning Strategies on Memorization

SFT improves original-task accuracy but increases HMI; PPO yields a better accuracy–HMI trade-off. Figure 5 shows differences across post-training strategies on Qwen-2.5-7B and Qwen-2.5-Coder-7B on BIGCODEBENCH. SFT consistently improves original-task accuracy. However, these gains are accompanied by significant increases in HMI. These trends indicate that, under our semantic-rewriting diagnostic, SFT can improve original-task accuracy while increasing brittleness to rewritten semantics. In contrast, PPO preserves baseline-level accuracy or better while reducing HMI. Overall, PPO yields a more favorable accuracy–HMI trade-off, keeping HMI near or below base levels while still maintaining competitive original-task accuracy. Table 5 further shows that later SFT checkpoints can yield small additional accuracy gains but disproportionately larger HMI increases, suggesting that HMI can serve as a lightweight checkpoint-selection diagnostic when validation accuracy alone is insufficient.

6 Conclusion and Future Work

In this paper, we study harmful memorization in code generation as failure under high similarity after task semantics change. We introduce code rewriting to construct superficially similar but semantically different task variants, and HMI to summarize whether models fail while retaining similarity to the original solution. Across MBPP+ and BigCodeBench, we find that harmful memorization does not monotonically increase with scale, concentrates in specific logic-edit types, and changes substantially under post-training: SFT can increase HMI despite improving original-task accuracy, while PPO yields a better accuracy–HMI trade-off. Future work should extend this diagnostic beyond function-level tasks to repository-level settings and multi-file semantic changes.

7 Limitations

Our diagnostic focuses on standalone function-level coding tasks and does not directly evaluate repository-level software engineering settings such as multi-file refactoring, long-horizon dependency reasoning, or SWE-Bench-style issue resolution. Our semantic rewrites are localized logic changes, which support controlled attribution but may not capture broader requirement changes in real-world codebases. HMI should also be interpreted together with original-task accuracy: low HMI does not necessarily imply strong coding ability when the model has low baseline accuracy. Finally, our rewriting pipeline uses an LLM judge for dataset validation, although candidate model scoring itself relies on executable tests and deterministic similarity measures.

8 Acknowledgements

Use of AI Assistance We used GPT-5 for generating candidate rewritten tasks and for validating prompt–solution consistency during dataset construction. Candidate model evaluation does not rely on GPT-5; it uses executable unit tests and deterministic similarity metrics. We also used GPT-5 for language polishing. All research ideas, experimental design, analysis, and conclusions were developed and verified by the authors.

9 Ethics Statement

Our *code rewriting* pipeline is guided by ethical principles to ensure responsible outcomes.

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

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

References

Meta AI. 2024. Introducing llama 4: Advancing multimodal intelligence.

Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2024. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–12. ACM.

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

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. *Preprint*, arXiv:2402.15938.

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

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

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, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv: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. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

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

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

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

Aneesh Pappu, Billy Porter, Ilia Shumailov, and Jamie Hayes. 2024. Measuring memorization in rlhf for code completion. *Preprint*, arXiv: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, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. *Preprint*, arXiv: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, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

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

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

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

Tabnine. 2024. Tabnine. https://www.tabnine.com. 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, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

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.

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

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

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

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 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 *code rewriting* variant with GPT-5. The system and user messages are shown as passed to the API.

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]
```

A.1 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 Code Rewriting Ground Truth Solution Validation

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 Training details regarding SFT/RL

C.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 4 GPUs (`n_gpus_per_node=2`). PPO rollouts used the `VLLM` backend; optimization used `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.

The completed template we fed into the LLM was:

```
instruction_prefix = "Please provide a self-contained Python script that
```

Table 6: 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`

Table 7: 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

```
    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.

C.1.1 Supervised Fine-Tuning (SFT)

Default learning rate was 1×10^{-5} for 20 epochs, with manual adjustments between 5×10^{-6} and 1×10^{-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.

Table 8: SFT hyperparameters

Parameter	Value
Epochs	20
Learning rate	Default 1×10^{-5}; tuned 5×10^{-6}–1×10^{-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

Table 9: 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)	1×10^{-5}
Learning rate (actor)	1×10^{-6}
Batch size	64
micro_batch_size_per_gpu	8
save_freq	5
Checkpoint selection	Highest reward on validset

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.

C.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 1×10^{-5} for the critic and 1×10^{-6} for the 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.
```

D Evolved-Task Generation (GPT-5)

- **API version**: `gpt-5-2025-08-07`.
- **Prompt template**: shown in Appendix A.
- **Parameters**: temperature: default; top-p: default; max-tokens 1080.
- **Post-processing**: regex clean-up.
- **Budget**: 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.

E Stability of RAD and HMI under low baseline accuracy

The Relative Accuracy Drop (RAD) measures the *fractional* decrease in Pass@1 between the original and rewritten task sets. Although this captures the intended notion of rewriting-sensitivity, it becomes unstable when the baseline accuracy $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$ is extremely low: even a small absolute change can inflate the relative ratio and consequently HMI (defined as the product of similarity and RAD).

To ensure that RAD and HMI reflect meaningful performance changes rather than numerical instability, we compute these metrics only when two conditions hold.

(1) Sufficient number of correctly solved original tasks. For a benchmark with N tasks and baseline accuracy $p = \mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$, the model solves $k = Np$ tasks. We require

$$k \geq M = 50,$$

which keeps the variance of the empirical Pass@1 sufficiently small under a binomial model. For instance, MBPP+ ($N = 378$) requires $p >= 13.2\%$, resulting in a 95% confidence interval of roughly ± 3–4 percentage points; BigCodeBench ($N = 1{,}140$) requires $p >= 4.4\%$ with an even tighter interval. These uncertainties are far smaller than the 5–20 point drops typically observed in our memorization analyses.

(2) A minimum competence threshold on baseline accuracy. We additionally require

$$\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}}) \geq 10\%,$$

which prevents the denominator in RAD from becoming so small that modest absolute changes (e.g., a 5-point drop) produce disproportionally large relative decreases (e.g., $> 50\%$).

Application of the threshold. When either condition is violated, we still report $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$ and $\mathrm{Acc}(\mathcal{T}_{\mathrm{rew}})$ for transparency, but we mark RAD and HMI as *"N/A (below competence threshold)"* and exclude them from cross-model comparisons. All analyses and figures in the main paper adhere to this stability rule.

These criteria ensure that large RAD values reflect genuine difficulty introduced by semantic rewriting rather than artifacts of low baseline accuracy, and that HMI remains a reliable indicator of harmful memorization behaviour.

F Analysis by Logic Change Type

Logic change types. To better understand *which* semantic edits drive memorization, we categorize each semantics-altering rewrite into one of the following nine logic change types:

- **Component Swap:** Swapping a function, class, or method while keeping the surrounding structure fixed (e.g., `sum` → `max`, `np.std` → `np.var`).
- **Data Transformation:** Transforming the data *before* applying the main logic (e.g., `np.log(data)`, `data.T`, `data.pct_change()`, tokenization or normalization steps).
- **Constraint Change:** Modifying rules, parameters, or logical conditions (e.g., loop stride in `range(..., 2)`, bounds in inequalities, regex patterns, or SQL `UNIQUE` constraints).
- **Logic Reversal:** Inverting the decision or ranking logic (e.g., `max` → `min`, `sorted(...)` → `sorted(..., reverse=True)`, swapping the roles of `x` and `y` in comparisons).
- **Operator Change:** Swapping a single arithmetic or logical operator (e.g., `^` → `+`, `==` → `!=`).
- **Operation Order Swap:** Reordering two or more operations with the same components (e.g., applying `blur` then `gray` vs. `gray` then `blur`).
- **Constant Swap:** Changing hard-coded constants or literal values (e.g., switching between `lowercase` and `UPPERCASE`, or adjusting thresholds).
- **Workflow Modification:** Structural changes to the overall pipeline, such as inserting or removing processing stages (e.g., adding an extra `zlib` compression step or altering nested outputs).

Logic Change Type	BigCodeBench #	BigCodeBench %	MBPP+ #	MBPP+ %
Total	1140	100.00%	378	100.00%
Component Swap	393	34.47%	52	13.76%
Data Transformation	385	33.77%	58	15.34%
Constraint Change	185	16.23%	86	22.75%
Logic Reversal	50	4.39%	68	17.99%
Workflow Modification	50	4.39%	10	2.65%
Constant Swap	37	3.25%	36	9.52%
Operation Order Swap	20	1.75%	12	3.17%
Operator Change	18	1.58%	36	9.52%
Other	2	0.18%	20	5.29%

Table 10: Code Rewriting logic change type statistics for `bigcodebench` and `mbppplus`.

- **Other:** Any semantics-altering edit that does not clearly fall into the above categories.

Coverage across benchmarks. Table 10 reports the distribution of logic change types across BIGCODEBENCH and MBPP+. On BIGCODEBENCH, the changes are dominated by **Component Swap** (34.5%) and **Data Transformation** (33.8%), followed by **Constraint Change** (16.2%), with the remaining types each accounting for less than 5% of all instances. On MBPP+, the distribution is more balanced: **Constraint Change** (22.8%), **Logic Reversal** (18.0%), **Data Transformation** (15.3%), **Component Swap** (13.8%), and **Constant/Operator Swap** (9–10%) all contribute substantially. To avoid unstable estimates, all per-type statistics in this appendix exclude logic-change categories with fewer than 50 samples in a given dataset.

Per-type RAD on MBPP+. Table 11 shows accuracy on original tasks $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$, rewritten tasks $\mathrm{Acc}(\mathcal{T}_{\mathrm{rew}})$, and the resulting accuracy drop RAD (all for semantics-altering rewrites, i.e., $\mathrm{RAD}_{\mathrm{rew}}$) for four Qwen-2.5 variants on MBPP+. Several patterns emerge:

- For both Instruct and Coder families, **Constraint Change** exhibits the largest RAD at both 0.5B and 14B, indicating that models are particularly brittle when loop bounds, thresholds, or other constraints are modified.
- **Data Transformation** also induces substantial drops (25–35% at 0.5B; 7–14% at 14B), showing the difficulty of tracking changes to data preprocessing or representation.
- **Component Swap** has comparatively *small* RAD and becomes almost benign at 14B, while **Logic Reversal** transitions from a sizeable drop at 0.5B to nearly no drop (or even slight improvements) at 14B.

These results show that, on the simpler MBPP+ benchmark, harmful memorization is not uniform across edit types: scaling primarily reduces vulnerability to Constraint Change and Data Transformation, while simple component/logic swaps become much easier for larger models to handle.

Per-type RAD on BigCodeBench. Table 12 reports the same statistics for BIGCODEBENCH. Compared to MBPP+, all major logic-change types now induce much larger drops:

- At 0.5B, **Constraint Change**, **Data Transformation**, **Logic Reversal**, and **Workflow Modification** all yield extremely high RAD (often above 70% for Qwen-2.5-0.5B-Instruct), confirming that models struggle when rewrites modify global data flow or constraints on this more challenging benchmark.
- Scaling to 14B substantially improves $\mathrm{Acc}(\mathcal{T}_{\mathrm{ori}})$, but RAD *remains high* for Data Transformation, Constraint Change, and Workflow Modification, with per-type drops still in the 30–45% range.

Thus, even at larger scales, BIGCODEBENCH exposes persistent brittleness when the rewrite changes how data are transformed, rather than merely swapping local components.

G 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.

Logic Change Type	qwen-2.5-coder-0.5b-instruct				qwen-2.5-coder-14b-instruct			
	Total	Acc($\mathcal{T}_{\text{ori}}$) (↑)	Acc($\mathcal{T}_{\text{rew}}$) (↑)	RAD (↓)	Total	Acc($\mathcal{T}_{\text{ori}}$) (↑)	Acc($\mathcal{T}_{\text{rew}}$) (↑)	RAD (↓)
Component Swap	52	48.1%	44.2%	8.0%	52	75.0%	71.2%	5.1%
Constraint Change	86	46.5%	30.2%	35.0%	86	73.3%	54.7%	25.4%
Data Transformation	58	55.2%	41.4%	25.0%	58	72.4%	62.1%	14.3%
Logic Reversal	68	42.6%	35.3%	17.2%	68	72.1%	76.5%	0.0%
HMI (all nine types)	–	–	–	**0.0615**	–	–	–	**0.0313**
	qwen-2.5-0.5b-instruct				qwen-2.5-14b-instruct			
Component Swap	52	44.2%	40.4%	8.7%	52	75.0%	71.2%	5.1%
Constraint Change	86	39.5%	24.4%	38.2%	86	64.0%	48.8%	23.6%
Data Transformation	58	50.0%	32.8%	34.5%	58	70.7%	65.5%	7.3%
Logic Reversal	68	39.7%	26.5%	33.3%	68	72.1%	80.9%	0.0%
HMI (all nine types)	–	–	–	**0.0722**	–	–	–	**0.0113**

Table 11: Comparison of logic change type performance between four Qwen-2.5 variants on `MBPP+`. We omit logic change types with fewer than 50 samples to avoid statistical bias. RAD highlighted in red denotes the highest drop.

Logic Change Type	qwen-2.5-coder-0.5b-instruct				qwen-2.5-coder-14b-instruct			
	Total	Acc($\mathcal{T}_{\text{ori}}$) (↑)	Acc($\mathcal{T}_{\text{rew}}$) (↑)	RAD (↓)	Total	Acc($\mathcal{T}_{\text{ori}}$) (↑)	Acc($\mathcal{T}_{\text{rew}}$) (↑)	RAD (↓)
Component Swap	393	13.5%	6.4%	52.8%	393	46.8%	29.3%	37.5%
Constraint Change	185	8.6%	4.9%	43.8%	185	50.8%	33.5%	34.0%
Data Transformation	385	10.6%	3.4%	68.3%	385	46.8%	26.8%	42.8%
Logic Reversal	50	8.0%	6.0%	25.0%	50	46.0%	34.0%	26.1%
Workflow Modification	50	10.0%	6.0%	40.0%	50	42.0%	24.0%	42.9%
HMI (all nine types)	–	–	–	**0.1778**	–	–	–	**0.1237**
	qwen-2.5-0.5b-instruct				qwen-2.5-14b-instruct			
Logic Change Type	Total	Acc($\mathcal{T}_{\text{ori}}$) (↑)	Acc($\mathcal{T}_{\text{rew}}$) (↑)	RAD (↓)	Total	Acc($\mathcal{T}_{\text{ori}}$) (↑)	Acc($\mathcal{T}_{\text{rew}}$) (↑)	RAD (↓)
Component Swap	393	12.0%	4.8%	59.6%	393	38.2%	26.7%	30.0%
Constraint Change	185	11.9%	2.7%	77.3%	185	40.5%	27.0%	33.3%
Data Transformation	385	8.3%	2.1%	75.0%	385	39.7%	21.8%	45.1%
Logic Reversal	50	8.0%	2.0%	75.0%	50	44.0%	28.0%	36.4%
Workflow Modification	50	8.0%	2.0%	75.0%	50	30.0%	18.0%	40.0%
HMI (all nine types)	–	–	–	**0.1740**	–	–	–	**0.0841**

Table 12: Comparison of logic change type performance between four Qwen-2.5 variants on `BigCodeBench`. We omit logic change types with fewer than 50 samples to avoid statistical bias. RAD highlighted in red denotes the highest drop.

```
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 6: 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 7: 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 8: 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 9: 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 10: 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 11: 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 12: 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 13: 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 14: 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 15: Example of Task-868 from BigCodeBench generated from Qwen2.5-Coder-32B-Instruct that PASSED in original but FAILED in code_rewriting.