# Duplicate-Aware Controlled Code Generation: Enhancing Copyright Protection with Targeted Reordering Beam Search in LLMs

**Anonymous ACL submission**

## Abstract

The increasing integration of large language models (LLMs) in code generation has raised critical copyright concerns, particularly regarding the verbatim repetition of copyrighted code. To address this challenge, we propose a novel task: Duplicate-Aware Controlled Code Generation (DACCG), which aims to mitigate verbatim repetition while preserving the quality of generated code. To this end, we introduce Targeted Reordering Beam Search (TRBS), a plug-and-play decoding method that dynamically reorders beam candidates to reduce direct copying. TRBS leverages the FM-index for efficient substring detection and employs a spike-entropy-based protection mechanism to safeguard structural anchors critical to code coherence. Experimental results on a multi-language code generation benchmark demonstrate that TRBS effectively reduces verbatim repetition while maintaining functional adequacy. Our research represents a pioneering effort in code copyright protection from the model user's perspective, offering novel insights into responsible code generation practices.[1]

## 1 Introduction

The legality of training on copyrighted data is context-dependent (David M. McIntosh, 2024) and is often justified under exceptions such as "fair use" in non-commercial research and educational settings (Klosek, 2024). However, the risk of infringement becomes more pronounced during inference. If a model generates content that substantially duplicates copyrighted material, it will likely constitute infringement, even under "fair use" (Ekene Chuks-Okeke, 2024; Scott M. Douglass, 2024).In most legal systems, both source code and object code are classified as literary works and receive equal protection under copyright law (WIPO, a,b).

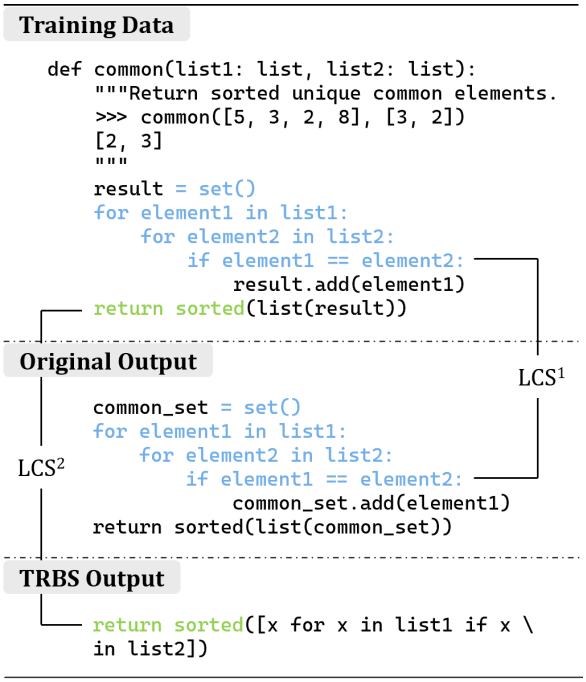


Figure 1: Illustration of the effect of TRBS on Llama3. $LCS^1$ denotes the longest common subsequence (LCS) between the training data and the model's original output, while $LCS^2$ refers to the LCS between the training data and the model's output with TRBS.

Research has shown that LLMs exhibit a tendency to reproduce verbatim during generation (Karamolegkou et al., 2023), directly copying portions of their training data. Karamolegkou et al. found that models with parameters smaller than 60B tend to reproduce an average of 50 words from literary works using simple prompting strategies. Such verbatim repetition poses significant copyright risks for model owners.

Due to the widespread application of LLMs in code generation, copyright protection for code-related data faces particularly complex challenges. For example, GitHub Copilot has been found to generate substantial segments of copied code without attribution to the original licensed sources (Ronacher; Davis), leading to a lawsuit alleging

[1]Our models and code will be publicly accessible upon acceptance.

copyright infringement. Similarly, we observed that when fine-tuning datasets contain copyrighted material, verbatim repetition becomes evident. For example, as illustrated in Figure 1, the fine-tuned model's output included repeated training code segments with a length of 102 characters.

Therefore, to protect the legitimate rights of copyright owners and reduce the risk of infringement for code generation, we propose a novel task: Duplicate-Aware Controlled Code Generation (DACCG). The objectives of DACCG can be categorized into two key dimensions:

DIMENSION 1. ***Repetition Reduction:*** The primary goal of DACCG is to ensure that the generated code satisfies predefined constraints on verbatim repetition with proprietary code files. The proprietary code files refer to copyright-protected materials used during model training.

DIMENSION 2. ***Quality Preservation:*** Beyond minimizing repetition, it is crucial to maintain the fluency, usability, and functional adequacy of the generated code, ensuring that its overall quality remains uncompromised.

Correspondingly, we introduce Targeted Reordering Beam Search (TRBS), a plug-and-play decoding intervention method that requires no additional training and incurs minimal computational overhead.TRBS dynamically reorders beam candidates at each decoding step to reduce verbatim repetition. To detect potential verbatim sequences, TRBS leverages the FM-index (Ferragina and Manzini, 2000), enabling efficient substring search and localization with linear space complexity and search time dependent on substring length rather than corpus size. Once potential verbatim sequences are identified, TRBS attempts to adjust beam scores by swapping them with semantically equivalent alternatives. To safeguard the structural anchors within beams—critical nodes that ensure code generation quality by preserving contextual coherence and integrity - we introduce a protection mechanism based on spike entropy dynamics. This mechanism detects structural anchors based on spike entropy (Strong et al., 1996) and strictly prevents replacement operations at these positions. These anchor points exhibit vulnerability characteristics in code generation, where even the substitution of seemingly non-critical tokens may trigger cascading effects, potentially driving subsequent generations into syntactically or semantically invalid regions. We argue that low spike entropy regions function as contextual dependency amplifiers in the generation process, where minor local modifications can lead to disproportionate and nonlinear impacts due to score accumulation, ultimately affecting code correctness and consistency.

To validate the effectiveness of our approach, we conducted extensive experiments on a code generation benchmark that includes five programming languages. Experimental results demonstrate that TRBS achieves superior performance in repetition reduction and generation quality compared to baseline methods. Notably, our method offers flexible control over the accuracy-repetition trade-off through simple hyperparameter tuning. The contributions of this paper are as follows:

- We define a novel task, DACCG, which reduces verbatim repetition in code generation while maintaining output quality. To the best of our knowledge, this is the first study to explore and mitigate copyright infringement risk in code generation from the perspective of model users.
- We develop TRBS, a beam search modification that enables plug-and-play repetition control without requiring additional computational overhead.
- We introduce a spike-entropy-based mechanism that ensures code generation quality by protecting structural anchors.

## 2 Background

### 2.1 Copyright Infringement Concern

As LLMs demonstrate the ability to memorize and generate verbatim segments from their training data, concerns about copyright infringement have gained increasing attention. Carlini et al. first demonstrated that language models, such as GPT-2, are susceptible to data extraction attacks, where adversaries can recover sensitive information from the model's outputs. Chang et al. explored LLMs' memorization of copyrighted books, focusing on cloze tasks, but did not consider whether these models might verbatim reproduce such texts. Karamolegkou et al. were among the first to systematically investigate copyright violations in LLMs, focusing on how models verbatim memorize and regenerate copyrighted works. Their findings show that LLMs tend to memorize and reproduce substantial portions of copyrighted content, particularly for widely popular materials.

This phenomenon poses substantial legal challenges as current copyright frameworks—while allowing limited reproduction under specific conditions—impose critical constraints. For instance, the

U.S. Fair Use doctrine permits 300-word verbatim quotations for purposes like criticism or education (Office), whereas EU Directive 2001/29/EC mandates quotations be strictly proportionate to specific purposes under fair practice (EU).

### 2.2 Controlled Code Generation

With the growing use of LLM-based code generation, recent studies have explored controlled code generation to enhance adaptability and reliability, guiding LLMs to meet specific constraints. For structured domain-specific languages (DSLs), Pimparkhede et al. proposed DocCGen, a two-stage framework that reduces syntactic and semantic errors. In industrial automation, Koziolek and Koziolek introduced a multimodal approach that integrates visual recognition and domain knowledge for control logic generation. Reliability-constrained methods like SVEN(He and Vechev, 2023) use learnable property-specific vectors to enforce security compliance without modifying model weights. While prior work focuses on structure, safety, and security, we address a new dimension: copyright legality. We introduce DACCG, the first systematic effort to explore and mitigate copyright infringement risks in LLM-generated code from the perspective of model users, bridging a critical gap in responsible code generation.

### 2.3 Decoding-time Intervention on Controllable Generation

Controllable Generation involves several key methods(Keskar et al., 2019; Zhou et al., 2023; Upadhyay et al., 2022; Shin et al., 2020; Subramani et al., 2022; Zhong et al., 2024). Among these, decoding-time intervention methods have received considerable attention, which adjusts LLMs' output logits during decoding to steer text generation towards desired attributes. It can be grouped into five types based on knowledge injection techniques (Liang et al., 2024). Classifier guidance methods use external classifiers, such as reward models or neural networks, during decoding to adjust the model's output and control specific text attributes (Deng and Raffel, 2023; Mudgal et al., 2024; Zheng et al., 2023a). Class-Conditioned Language Models (CC-LMs) leverage pre-trained or fine-tuned models to guide text generation based on predefined attributes like sentiment or topic, though using them directly may yield suboptimal results (Zhong et al., 2023). To enhance control, the logits from CC-LMs are utilized as guidance during the decoding process. Self-feedback guidance methods utilize the model's internal knowledge to adjust generated text during decoding, ensuring alignment with desired attributes despite insufficient prompts or constraints (Pei et al., 2023; Zhong et al., 2024). Energy-based models (EBMs) optimize the generation process by adjusting energy values to meet specific constraints, facilitating the balance of multiple attributes (Guo et al., 2024; Yu et al., 2024). Finally, external knowledge guidance techniques incorporate information from external knowledge bases or retrieval systems to improve text consistency and alignment with desired attributes, typically through semantic guidance or knowledge retrieval (Han et al., 2024).

## 3 Task: DACCG

We propose a novel task named Duplicate-Aware Controlled Code Generation, which aims to reduce verbatim repetition from proprietary code files in LLM outputs while preserving generation quality.

Since most legal frameworks define infringement boundaries based on the max length of copied sequences (Office; EU), we adopt the Longest Common Subsequence (LCS) length as the primary metric for measuring repetition. Given a reference code snippet $R$ and a generated output $G$, the LCS length is defined as $\max\{|C| \mid C \sqsubseteq R \cap C \sqsubseteq G\}$. Here, the $\sqsubseteq$ denotes the subsequence relationship. It should be noted that the DACCG task focuses on code copyright infringement rather than code plagiarism. These two concepts bear fundamental legal distinctions. Copyright infringement constitutes an unlawful act characterized by substantial similarity at the literal expression level, typically measured by the LCS length (Karamolegkou et al., 2023). In contrast, plagiarism primarily relates to ethical and academic integrity concerns, where similarity is generally assessed through structural resemblances (Schleimer et al., 2003).

For code generation, quality can be directly measured by execution accuracy, which measures functional correctness by executing generated code against ground-truth test cases.

From the perspective of intervention stages, potential solutions for DACCG can be categorized into three classes: (1) Preprocessing interventions, such as data rewriting or augmentation to minimize overlap from proprietary code files; (2) Decoding interventions, which involve constraint-based modifications during decoding; and (3) Postprocessing

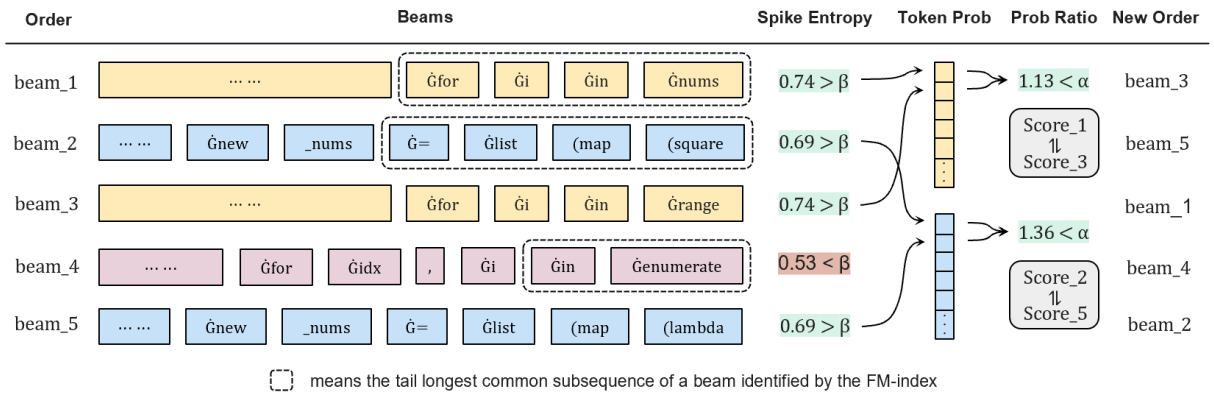


Figure 2: How TRBS works. The uniform color of the beams signifies that they originate from the same parent beam. The red conditional expression indicates that a beam has satisfied the given criterion, while green signifies the contrary.

interventions, where generated outputs are refined or adjusted to reduce repetition after generation.

## 4 Method

Given proprietary code files $P$, TRBS begins by initializing the FM-index based on the token id sequence $S$ of the files. This process involves generating the Burrows-Wheeler Transform (BWT) (Burrows et al., 1994) and suffix array, followed by constructing auxiliary arrays to record the global ranking of characters and their occurrence counts in BWT prefixes. Once the FM-index is prepared, TRBS initializes the key variables required for beam search and enters the iterative token generation process.

During each generation epoch, TRBS operates in two main phases. The first phase follows the standard beam search procedure, performing forward computation, token selection, and beam hypothesis updates to generate new input ids for the next step. The second phase, constituting the core innovation of our method, dynamically adjusts the ranking of beam candidates. This adjustment is based on their trajectory in reproducing verbatim repetition while maintaining generation quality, as demonstrated in Figure 2.

To detect potential verbatim sequence, TRBS maintains a mask matrix $M$ that tracks the Tail Longest Common Subsequence (TLCS) between each beam candidate $b_i$ in the candidate set $C = \{b_1, b_2, \ldots, b_n\}$ and the sequence $S$. The TLCS represents the longest subsequence shared between the suffix of a beam candidate and the proprietary code files. Formally, the TLCS for a beam $b_i$is defined as:

$$TLCS(b_i, S) = \arg\max_{t_{ik}} \{k \mid t_{ik} \sqsubseteq S\},$$

where $t_{ik}$ denotes the $k$-length suffix subsequence of beam $b_i$. The mask matrix $M \in \{0,1\}^{(n \cdot N) \times L}$ is updated iteratively during the generation process, where $L$ is the current beam length, $n$ is the beam num, and $N$ is the batch size. Each element $M[i,j]$ indicates whether the $j$-th token of beam $b_i$ belongs to its TLCS with respect to $S$.

During the reordering phase, TRBS uses the FM-index to perform pattern matching and determine whether the newly generated tokens, when appended to a beam candidate, result in a sequence that exists in $S$. If a match is found, the corresponding entry in the mask matrix is set to 1; otherwise, it is set to 0. For beams with a non-zero TLCS length, TRBS attempts to find a replacement candidate that originates from the same parent beam and satisfies two key conditions: (1) the replacement candidate has a TLCS length of zero, indicating it is not reproducing the proprietary code files, and (2) the replacement candidate is semantically substitutable, meaning the substitution does not significantly degrade the quality of the generation.

Semantic substitutability is evaluated through a two-tiered criterion. At the token level, the probability ratio between the original token $t_m$ and the candidate replacement token $t_n$ in the model's logits is used to filter out low-quality substitutions. Specifically, a replacement candidate $b_m$ will be rejected if the ratio $p_m/p_n$ exceeds a predefined threshold $\alpha$.

At the contextual level, TRBS employs spike entropy to quantify the substitutability of beam positions and identify structural anchors. The spike entropy for beam $b_i$ is calculated as:

$$SpikeEntropy = \sum_{m=1}^{D} \frac{P_i(m)}{1 + z \cdot P_i(m)},$$

where $z$ is a scaling constant and $P_i(m)$ denotes the probability distribution of the logits for beam $b_i$. Positions with spike entropy values below a threshold $\beta$ are identified as structural anchors – critical nodes where token substitutions are prohibited to prevent cascading errors that could disrupt syntactic or semantic coherence.

Both $\alpha$ and $\beta$ are empirically determined hyperparameters that control the trade-off between repetition reduction and generation quality. Through systematic adjustment of these thresholds, users can flexibly balance copyright compliance requirements with code utility preservation.

# 5 Experiment

## 5.1 Experimental Setup

**Scenario and Dataset** To simulate a scenario where the training data contains both copyrighted and publicly available data, we tune LLMs on HumanEval-X (Zheng et al., 2023b) and DS-1000 (Lai et al., 2023), designating HumanEval-X as the proprietary code files. During evaluation, we assess the LCS length and execution accuracy of the generated code on HumanEval-X. For detailed information on these datasets, please refer to the appendix C.

**Model Configuration** We select Llama3-instruct-8B (AI@Meta, 2024) as the primary LLM to validate the effectiveness of TRBS. Our experiments employ Low-Rank Adaptation (LoRA) (Hu et al., 2021) to tune LLMs with rank 8 and alpha 16, training for 10 epochs with a batch size of 4 and a learning rate of 1e-4. To ensure reproducibility, all models utilize greedy decoding with maximum new tokens constrained to 256.

## 5.2 Main Result

We present the performance comparison in Figure 3 to validate the effectiveness of TRBS. The detailed experimental data can be found in Appendix B. Specifically, we conduct two key investigations. First, we introduce two straightforward baseline methods for the DACCG task:

• Training Data Rewrite: A preprocessing intervention method where proprietary code files are rewritten using an LLM with DACCG objectives before being used as training data.

• Output Regeneration: A postprocessing intervention method where the model's initial output is appended to the prompt to guide a second-generation attempt under DACCG objectives.

Second, we perform an ablation study by removing the structural anchor protection mechanism to examine its role within TRBS. Our findings indicate the following:

**TRBS enables more flexible control over the accuracy-repetition trade-off compared to baselines.** Adjusting $\alpha$ from low to high results in a reduction of LCS length at the cost of some accuracy loss. This allows users to achieve their desired balance between high LCS length with high accuracy and low LCS length with low accuracy. Moreover, TRBS exhibits stable LCS length and accuracy variations across different threshold settings, demonstrating its strong controllability. In contrast, the baseline methods show more erratic performance in the accuracy-repetition trade-off.

**TRBS outperforms baseline methods in DACCG tasks.** When analyzing both LCS length and accuracy, the polynomial fit curve of TRBS consistently appears in the upper-right region compared to the baselines across all five datasets. This indicates that for similar LCS length values, TRBS achieves higher accuracy than the baselines, and for similar accuracy values, TRBS achieves lower LCS length. Specifically, in the Python and Java datasets, TRBS surpasses Training Data Rewrite in accuracy while achieving a lower LCS length at $\alpha = 4$ and $\alpha = 5$, respectively. In the JavaScript and CPP datasets, TRBS at $\alpha = 5$ achieves a slightly lower LCS length than Training Data Rewrite but maintains significantly higher accuracy. Additionally, across all five datasets, there exists at least one setting where TRBS surpasses Output Regeneration in accuracy while maintaining a lower LCS length.

**Structural anchor protection is effective.** The polynomial fit curve for TRBS without structural anchor protection consistently falls in the lower-left region across all five datasets. Examining specific data points, and removing structural anchor protection results in a slight increase in repetition but a substantial drop in accuracy. This suggests that structural anchor protection effectively filters out replacements that significantly degrade generation quality. Notably, the lower the $\alpha$, the less pronounced the impact of structural anchor protection on TRBS performance. This is because a lower $\alpha$ imposes stricter constraints on token-level semantic substitutability, reducing the number of potential replacements during reordering. Consequently, even with structural anchor protection, the actual reordering operations remain largely unchanged.

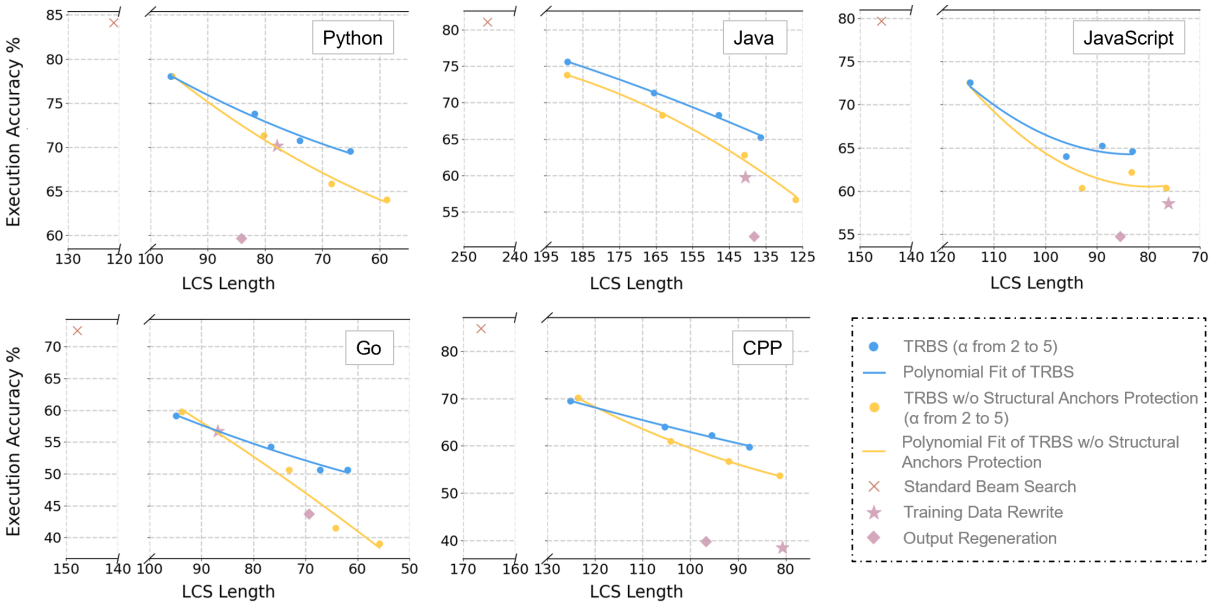


Figure 3: Comparative experiments and ablation study for structure anchor protection on HumanEval-X with Llama3-instruct-8B. In each subplot, the data points for TRBS from left to right correspond to $\alpha$ values ranging from 2 to 5, with the beam num set to 5 and $\beta$ set to 0.55.

### 5.3 Impact of $\beta$ and Its Interaction with $\alpha$

Section 5.2 has demonstrated that adjusting $\alpha$ enables a trade-off between repetition and accuracy. To further explore the impact of $\beta$ on generation performance, the interaction between $\alpha$ and $\beta$, and effective tuning strategies for both, we present in Table 1 the effect of varying $\beta$ under different $\alpha$ values. Figures 4 depict the distribution of token probability ratio and the distribution of logits spike entropy for all potential replacements in the Python dataset. Our findings are as follows.

First, $\beta$ also facilitates a balance between repetition and accuracy to some extent. As $\beta$ increases, the LCS length rises, while the accuracy decreases. This trend indicates that the criteria for determining contextual semantic substitutability become more stringent with a higher $\beta$.

Second, larger $\alpha$ values amplify the sensitivity of LCS length and accuracy to changes in $\beta$. As $\alpha$ increases, the token-level semantic substitutability criteria become more relaxed, making $\beta$ more influential in determining the number of replacements. This suggests that $\alpha$ sets the baseline for repetition reduction, while higher $\alpha$ values accentuate the role of $\beta$ in the beam reordering process.

Finally, we observe that the sensitivity of LCS length and accuracy to $\alpha$ and $\beta$ varies across different equal-length intervals. As shown in Table 1, the magnitude of LCS length and accuracy variations decreases as $\alpha$ increases in equal steps. Similarly, when $\beta$ increases from 0.55 to 0.60, LCS length and accuracy exhibit greater sensitivity compared to the change from 0.50 to 0.55. This phenomenon is dictated by the distribution of token probability ratio and the distribution of logits spike entropy, which tend to be concentrated for a given model and domain. As illustrated in Figures 4, the token probability ratio of potential replacements in TRBS is concentrated in the 0.50–0.55 range, while the logits spike entropy predominantly falls within the 0.55–0.60 range. Consequently, minor adjustments to $\alpha$ and $\beta$ within these ranges significantly influence the filtering of replacement.

These conclusions suggest that by tuning $\alpha$ and $\beta$ within specific ranges, one can flexibly balance repetition reduction and generation quality. Moreover, the interplay between $\alpha$ and $\beta$ underscores the importance of jointly optimizing these hyperparameters to precisely adapt TRBS for specific domains and tasks.

### 5.4 Parameter Analysis of Beam Num

We present the TRBS performance when varying beam num in Table 2. As the beam num increases, the overall LCS length exhibits a decreasing trend, accompanied by a slight decline in accuracy. This is because a larger beam num provides more candidates for beam substitution, thereby increasing the likelihood of replacements occurring. However, when beam num exceeds 15, the rate of LCS length reduction gradually diminishes, while the decline

| $\beta$ | Python | | Java | | JavaScript | | Go | | CPP | |
|---|---|---|---|---|---|---|---|---|---|---|
| | LCS | ACC | LCS | ACC | LCS | ACC | LCS | ACC | LCS | ACC |
| $\alpha = 2$ | | | | | | | | | | |
| *0.50* | 96.20 | 78.05 | 189.27 | 73.78 | 114.69 | 72.56 | 93.79 | 59.76 | 123.55 | 70.12 |
| *0.55* | 96.46 | 78.05 | 189.20 | 75.61 | 114.60 | 72.56 | 94.86 | 59.15 | 125.08 | 69.51 |
| *0.60* | 97.85 | 78.66 | 191.91 | 77.44 | 116.33 | 75.00 | 98.54 | 62.20 | 128.34 | 71.95 |
| $\alpha = 3$ | | | | | | | | | | |
| *0.50* | 80.20 | 71.34 | 163.26 | 68.29 | 92.87 | 60.37 | 73.13 | 50.61 | 104.15 | 60.98 |
| *0.55* | 81.79 | 73.78 | 165.48 | 71.34 | 95.95 | 64.02 | 76.65 | 54.27 | 105.34 | 64.02 |
| *0.60* | 89.57 | 76.83 | 170.30 | 75.00 | 103.32 | 70.12 | 82.15 | 58.54 | 112.73 | 69.51 |
| $\alpha = 4$ | | | | | | | | | | |
| *0.50* | 68.39 | 65.85 | 140.88 | 62.80 | 83.31 | 62.20 | 64.20 | 41.46 | 91.97 | 56.71 |
| *0.55* | 73.90 | 70.73 | 147.88 | 68.29 | 88.98 | 65.24 | 67.17 | 50.61 | 95.53 | 62.20 |
| *0.60* | 86.20 | 76.22 | 163.22 | 73.78 | 99.54 | 69.51 | 76.79 | 56.71 | 103.73 | 67.68 |
| $\alpha = 5$ | | | | | | | | | | |
| *0.50* | 58.74 | 64.02 | 126.90 | 56.71 | 76.52 | 60.37 | 55.79 | 39.02 | 81.32 | 53.66 |
| *0.55* | 65.10 | 69.51 | 136.45 | 65.24 | 83.10 | 64.63 | 61.93 | 50.61 | 87.67 | 59.76 |
| *0.60* | 80.95 | 76.22 | 158.64 | 73.17 | 96.26 | 70.12 | 75.15 | 57.32 | 98.41 | 65.85 |

Table 1: TRBS performance when varying $\beta$ under different $\alpha$ on HumanEval-X with Llama3-instruct-8B, with the beam num set to 5.

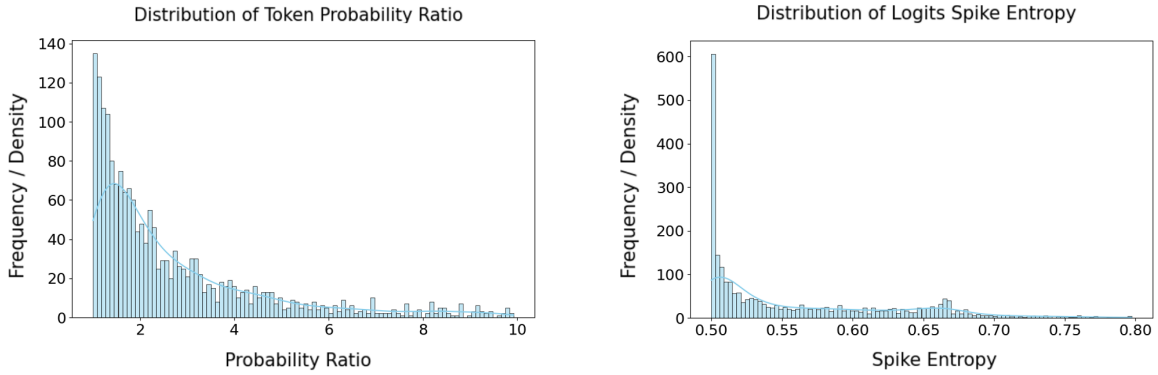


Figure 4: Distribution of token probability ratio and logits spike entropy in the Python dataset on Llama3-instruct-8B.

in accuracy becomes more pronounced, indicating diminishing returns in repetition control at larger beam num settings. Additionally, experiments reveal that different datasets exhibit varying sensitivities to changes in beam num, with the Go and CPP datasets being more sensitive. For example, when beam num increases from 5 to 25, the LCS length for the Go dataset drops from 67.17 to 63.37, whereas for the Java dataset, it only decreases slightly from 147.88 to 147.21. These results suggest that selecting an appropriate beam num value is crucial for achieving an optimal balance between repetition control and generation quality.

### 5.5 TRBS on Different LLMs

To further evaluate the performance of TRBS across different models, we conducted experiments on DeepSeek-R1-Distill-Llama-8B (et al., 2025) and Qwen2.5-7B-Instruct (Team, 2024). As shown in Figure 5, compared to Llama3, DeepSeek-R1-Distill-Llama tends to maintain higher accuracy at higher LCS length, whereas Qwen2.5 exhibits lower accuracy at lower LCS length. Despite these variations in model behavior, TRBS performs consistently well, surpassing both baseline methods across both models. Furthermore, the structure anchor protection mechanism proves effective across both models. Notably, on Qwen2.5-7B-Instruct, as $\alpha$ increases, structure anchor protection ensures that while LCS length decreases significantly, accuracy remains largely stable.

### 5.6 Modification Patterns of TRBS

To investigate how TRBS modifies code generation to reduce repetition, we analyzed the experimental results on the Python dataset and manually

| Beam Num | Python | | Java | | JavaScript | | Go | | CPP | |
|---|---|---|---|---|---|---|---|---|---|---|
| | LCS | ACC | LCS | ACC | LCS | ACC | LCS | ACC | LCS | ACC |
| *5* | 73.90 | 70.73 | 147.88 | 68.29 | 88.98 | 65.24 | 67.17 | 50.61 | 95.53 | 62.20 |
| *10* | 70.97 | 68.90 | 150.24 | 69.51 | 88.40 | 65.24 | 63.66 | 48.78 | 93.62 | 63.41 |
| *15* | 69.77 | 69.51 | 147.87 | 66.46 | 89.51 | 67.07 | 62.13 | 48.17 | 91.77 | 59.15 |
| *20* | 69.68 | 68.90 | 147.12 | 67.07 | 87.95 | 65.24 | 62.04 | 47.56 | 91.15 | 60.37 |
| *25* | 69.57 | 68.29 | 147.21 | 67.68 | 86.49 | 64.02 | 63.37 | 46.95 | 92.23 | 60.98 |

Table 2: TRBS performance when varying beam num on HumanEval-X with Llama3-instruct-8B, with $\alpha$ set to 4 and $\beta$ set to 0.55.

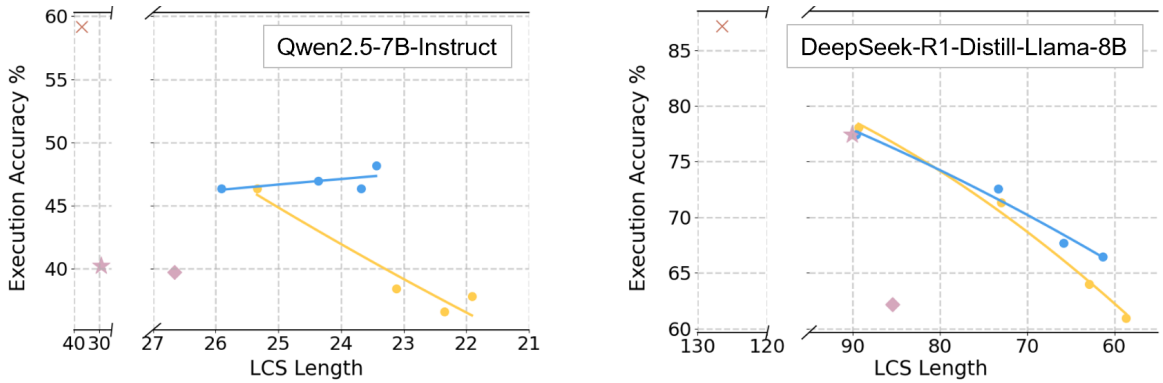


Figure 5: TRBS performance on the Python dataset with different LLMs. In each subplot, the data points for TRBS from left to right correspond to $\alpha$ values ranging from 2 to 5, with the beam num set to 5 and $\beta$ set to 0.55. The legend of this figure can be found in Figure 3, as they are the same.

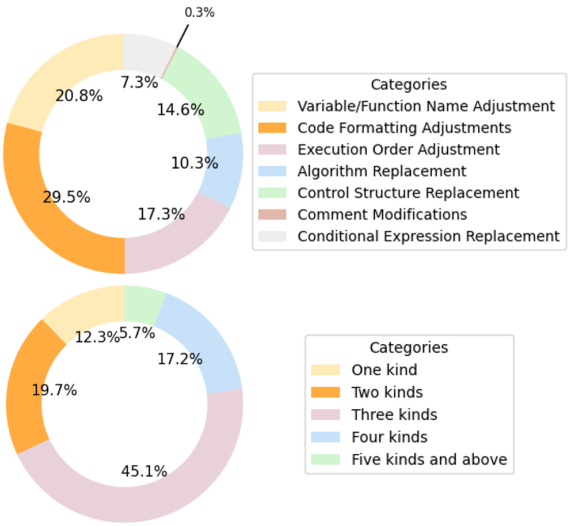


Figure 6: The frequency of each modification type and the distribution of the number of modification types per instance on the Python dataset.

categorized the modifications into seven distinct types. Figure 6 presents the frequency of each modification type and the distribution of the number of modification types per instance. The results indicate that Code Formatting Adjustments, Variable/Function Name Adjustment, Execution Order Adjustment, and Control Structure Replacement are the primary modification strategies, collectively accounting for 82.2% of all modifications. Moreover, modifications in a single instance often involve a combination of multiple types, with instances containing only one modification type comprising just 12.3% of the total.

## 6 Conclusion

In this paper, we introduced Duplicate-Aware Controlled Code Generation (DACCG), a novel task aimed at mitigating verbatim repetition in LLM-based code generation while preserving output quality. To address this challenge, we proposed Targeted Reordering Beam Search (TRBS), a plug-and-play decoding intervention that dynamically reorders beam candidates to minimize direct copying. TRBS efficiently detects potential verbatim sequences using the FM-index and employs a spike-entropy-based protection mechanism to maintain the structural integrity of generated code. Through extensive experiments on a multi-language code generation benchmark, we demonstrated that TRBS effectively reduces repetition while maintaining functional correctness. Our findings highlight the feasibility of controlling LLM output to reduce copyright risks without requiring additional training. This work provides a foundation for future research on controlled code generation and legal compliance in AI-assisted software development.

## Limitations

First, although TRBS eliminates the inference/training overhead characteristic of methods like Output Regeneration and Training Data Rewrite, it introduces computational costs through real-time substring searches. These operations, though optimized via FM-index's corpus-size-independent time complexity, create measurable processing overhead during beam verification. This represents a fundamental efficiency-reliability tradeoff: our approach maintains substantially lower operational costs than alternatives requiring complete output recomputation or linear-time corpus scans, yet may prove suboptimal for extreme low-latency applications. The architecture deliberately prioritizes verifiable text grounding while preserving deployable responsiveness for most practical use cases.

Second, while TRBS demonstrates technical effectiveness in reducing verbatim repetition at the algorithmic level, its legal implications remain complex. Current copyright frameworks primarily rely on the maximum consecutive matching substring to determine infringement, a standard that DACCG specifically targets in its technical design. However, this may not fully account for other legal principles, such as substantial similarity or the qualitative assessment of creative expression. The inherent complexity of copyright jurisprudence means that compliance with repetition thresholds at the technical level does not necessarily equate to the absence of infringing behavior.

## Ethics Statement

We take ethical considerations very seriously and strictly adhere to the ACL Ethics Policy. This study aims to reduce verbatim repetition in code generation to protect the legitimate rights of copyright holders while maintaining output quality. We introduce controlled modifications without restricting creative freedom, promoting responsible AI deployment rather than censorship. All pre-trained models and evaluation datasets used in this study are publicly available and widely adopted by researchers. Additionally, we acknowledge the potential risk of misuse and therefore call for the responsible use of our method to ensure ethical AI-generated content.

## References


AI@Meta. 2024. Llama 3 model card.

Michael Burrows, D J Wheeler D I G I T A L, Robert W. Taylor, David J. Wheeler, and David Wheeler. 1994. A block-sorting lossless data compression algorithm.

Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650. USENIX Association.

Kent Chang, Mackenzie Cramer, Sandeep Soni, and David Bamman. 2023. Speak, memory: An archaeology of books known to ChatGPT/GPT-4. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7312–7327, Singapore. Association for Computational Linguistics.

Yam Schaal David M. McIntosh, Georgina Jones Suzuki. 2024. Ai and the copyright liability overhang: A brief summary of the current state of ai-related copyright cases.

Tim Davis. @github copilot, with "public code" blocked, emits large chunks of my copyrighted code, with no attribution, no lgpl license.

Haikang Deng and Colin Raffel. 2023. Reward-augmented decoding: Efficient controlled text generation with a unidirectional reward model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11781–11791, Singapore. Association for Computational Linguistics.

Brenda Leong Ekene Chuks-Okeke, Natalie Linero. 2024. Generative ai and intellectual property: Copyright implications for ai inputs, outputs.

DeepSeek-AI et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

EU. Directive (eu) 2019/790 of the european parliament and of the council of 17 april 2019 on copyright and related rights in the digital single market and amending directives 96/9/ec and 2001/29/ec (text with eea relevance.).

P. Ferragina and G. Manzini. 2000. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398.

Xingang Guo, Fangxu Yu, Huan Zhang, Lianhui Qin, and Bin Hu. 2024. COLD-attack: Jailbreaking LLMs with stealthiness and controllability. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16974–17002. PMLR.

Chi Han, Jialiang Xu, Manling Li, Yi Fung, Chenkai Sun, Nan Jiang, Tarek Abdelzaher, and Heng Ji. 2024. Word embeddings are steers for language models. In

*Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16410–16430, Bangkok, Thailand. Association for Computational Linguistics.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879, New York, NY, USA. Association for Computing Machinery.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Antonia Karamolegkou, Jiaang Li, Li Zhou, and Anders Søgaard. 2023. Copyright violations and large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7403–7412, Singapore. Association for Computational Linguistics.

Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. Ctrl: A conditional transformer language model for controllable generation. *ArXiv*, abs/1909.05858.

Katherine Klosek. 2024. Training generative ai models on copyrighted works is fair use.

Heiko Koziolek and Anne Koziolek. 2024. Llm-based control code generation using image recognition. In *2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 38–45.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: a natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Xun Liang, Hanyu Wang, Yezhaohui Wang, Shichao Song, Jiawei Yang, Simin Niu, Jie Hu, Dan Liu, Shunyu Yao, Feiyu Xiong, and Zhiyu Li. 2024. Controllable text generation for large language models: A survey. *ArXiv*, abs/2408.12599.

Sidharth Mudgal, Jong Lee, Harish Ganapathy, YaGuang Li, Tao Wang, Yanping Huang, Zhifeng Chen, Heng-Tze Cheng, Michael Collins, Trevor Strohman, Jilin Chen, Alex Beutel, and Ahmad Beirami. 2024. Controlled decoding from language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

U.S. Copyright Office. Chapter 1: Subject matter and scope of copyright.

Jonathan Pei, Kevin Yang, and Dan Klein. 2023. PREADD: Prefix-adaptive decoding for controlled text generation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 10018–10037, Toronto, Canada. Association for Computational Linguistics.

Sameer Pimparkhede, Mehant Kammakomati, Srikanth G. Tamilselvam, Prince Kumar, Ashok Pon Kumar, and Pushpak Bhattacharyya. 2024. DocCGen: Document-based controlled code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18681–18697, Miami, Florida, USA. Association for Computational Linguistics.

Armin Ronacher. I don't want to say anything but that's not the right license mr copilot.).

Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD Conference*.

Dominic Rota Scott M. Douglass. 2024. The fast-moving race between gen-ai and copyright law.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4222–4235, Online. Association for Computational Linguistics.

S. Strong, Roland Koberle, Rob Steveninck, and William Bialek. 1996. Entropy and information in neural spike trains. *Physical Review Letters*, 80.

Nishant Subramani, Nivedita Suresh, and Matthew Peters. 2022. Extracting latent steering vectors from pretrained language models. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 566–581, Dublin, Ireland. Association for Computational Linguistics.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Bhargav Upadhyay, Akhilesh Sudhakar, and Arjun Maheswaran. 2022. Efficient reinforcement learning for unsupervised controlled text generation.

WIPO. a. Berne convention for the protection of literary and artistic works.

WIPO. b. U.s. copyright act, 17 u.s.c. §§ 101 et seq.

Sangwon Yu, Changmin Lee, Hojin Lee, and Sungroh Yoon. 2024. Controlled text generation for black-box language models via score-based progressive editor. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14215–14237, Bangkok, Thailand. Association for Computational Linguistics.

Carolina Zheng, Claudia Shi, Keyon Vafa, Amir Feder, and David Blei. 2023a. An invariant learning characterization of controlled text generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3186–3206, Toronto, Canada. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Qihuang Zhong, Liang Ding, Juhua Liu, Bo Du, and Dacheng Tao. 2024. ROSE doesn't do that: Boosting the safety of instruction-tuned large language models with reverse prompt contrastive decoding. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13721–13736, Bangkok, Thailand. Association for Computational Linguistics.

Tianqi Zhong, Quan Wang, Jingxuan Han, Yongdong Zhang, and Zhendong Mao. 2023. Air-decoding: Attribute distribution reconstruction for decoding-time controllable text generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8233–8248, Singapore. Association for Computational Linguistics.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Ethan Wilcox, Ryan Cotterell, and Mrinmaya Sachan. 2023. Controlled text generation with natural language instructions. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

## A Pseudo Code for TRBS

In this section, we present the pseudo code for TRBS in algorithm 1.

**Algorithm 1** Pseudo code of TRBS.

```
# max_new_tokens: maximum number of tokens to generate
# P: proprietary code files
# alpha: threshold for token-level semantic substitutability
# beta: threshold for contextual semantic substitutability

# Initialize FM-index with proprietary code files
S = tokenizer.encode(P)
fm_index = FM_index(S)

# Initialize beam search variables
beams = initialize_beams(batch_size, beam_num)
mask_matrix = initialize_mask_matrix(
    input_ids, beam_num, batch_size
)

for _ in range(max_new_tokens):
    # Phase 1: Standard beam search
    logits = model.forward(input_ids)
    next_tokens = select_tokens(logits, beams)
    beams = update_beams(beams, next_tokens)

    # Prepare for the next iteration
    input_ids = torch.cat(
        [input_ids, beams.next_tokens], dim=-1
    )

    # Phase 2: Targeted Reordering
    for b_idx in range(len(beams)):
        # Compute TLCS for the current beam
        last_tlcs = compute_tlcs(beams[b_idx], S, mask_matrix)

        # Update mask matrix based on the new token
        is_in_S = fm_index.pattern_match(
            last_tlcs + beams[b_idx][-1]
        )
        update_mask_matrix(mask_matrix, b_idx, is_in_S)

        # If TLCS is non-zero, find a replacement
        if is_in_S:
            # Calculate contextual substitutability
            spike_entropy = compute_spike_entropy(
                logits, b_idx
            )
            if spike_entropy < beta:
                continue

            # Find a suitable replacement beam
            min_prob_ratio = float("inf")
            replacement_idx = -1
            for other_b_idx in range(len(beams)):
                if other_b_idx ≠ b_idx and \
                beams[other_b_idx][:-1] == beams[b_idx][:-1]:
                    # Calculate token-level substitutability
                    prob_ratio = compute_probability_ratio(
                        logits, b_idx, other_b_idx
                    )
                    if prob_ratio < alpha and \
                    prob_ratio < min_prob_ratio:
                        min_prob_ratio = prob_ratio
                        replacement_idx = other_b_idx

            # Perform replacement if a candidate is found
            if replacement_idx ≥ 0:
                beams.swap_scores(b_idx, replacement_idx)

# Decode final output
sequence_outputs = beam_finalize(input_ids, beams)
```

## B Detailed Experimental Data

In this section, we present the detailed experimental data for the main result in Section 5.2 and the experiment between different models in Section 5.5. Specifically, Table 5 corresponds to Figure 1, while Tables 4 and 3 correspond to Figure 5.

## C Details of Datasets

### C.1 HumanEval-X

HumanEval-X is a multilingual extension of the HumanEval benchmark, designed to systematically evaluate multilingual code generation and translation capabilities. While HumanEval, like MBPP and APPS, consists solely of handcrafted Python programming problems, it cannot be directly applied to assess performance across multiple programming languages. To address this limitation, HumanEval-X extends HumanEval by manually translating each Python problem into four additional languages: C++, Java, JavaScript, and Go. Each problem-solution pair in HumanEval-X includes:

• task_id: A unique identifier for each problem, specifying the programming language and problem index (e.g., Java/0 represents the 0th problem in Java).

• declaration: The function declaration, includ-

| Method | LCS | ACC |
|---|---|---|
| QWEN2.5-7B<br>– Standard Beam Search | 20.20 | 40.85 |
| QWEN2.5-FINETUNE-7B<br>– Standard Beam Search | 36.98 | 59.15 |
| QWEN2.5-FINETUNE-7B<br>– Output Regeneration | 26.66 | 39.63 |
| QWEN2.5-FINETUNE-7B<br>– Training Data Rewrite | 29.14 | 40.24 |
| QWEN2.5-FINETUNE-7B<br>– TRBS w\o SAP ($\alpha = 2$) | 25.34 | 46.34 |
| QWEN2.5-FINETUNE-7B<br>– TRBS w\o SAP ($\alpha = 3$) | 23.12 | 38.41 |
| QWEN2.5-FINETUNE-7B<br>– TRBS w\o SAP ($\alpha = 4$) | 22.35 | 36.59 |
| QWEN2.5-FINETUNE-7B<br>– TRBS w\o SAP ($\alpha = 5$) | 21.91 | 37.80 |
| QWEN2.5-FINETUNE-7B<br>– TRBS ($\alpha = 2$) | 25.91 | 46.34 |
| QWEN2.5-FINETUNE-7B<br>– TRBS ($\alpha = 3$) | 24.36 | 46.95 |
| QWEN2.5-FINETUNE-7B<br>– TRBS ($\alpha = 4$) | 23.68 | 46.34 |
| QWEN2.5-FINETUNE-7B<br>– TRBS ($\alpha = 5$) | 23.44 | 48.17 |

Table 3: The detailed experimental data of comparative experiments and ablation study for structure anchor protection (SAP) on Python datasets with Qwen2.5-7B-Instruct.

| Method | LCS | ACC |
|---|---|---|
| DEEPSEEK-8B<br>– Standard Beam Search | 24.71 | 40.85 |
| DEEPSEEK-FINETUNE-8B<br>– Standard Beam Search | 126.55 | 87.20 |
| DEEPSEEK-FINETUNE-8B<br>– Output Regeneration | 85.46 | 62.20 |
| DEEPSEEK-FINETUNE-8B<br>– Training Data Rewrite | 90.06 | 77.44 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS w\o SAP ($\alpha = 2$) | 89.35 | 78.05 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS w\o SAP ($\alpha = 3$) | 73.01 | 71.34 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS w\o SAP ($\alpha = 4$) | 62.87 | 64.02 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS w\o SAP ($\alpha = 5$) | 58.66 | 60.98 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS ($\alpha = 2$) | 89.60 | 77.44 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS ($\alpha = 3$) | 73.34 | 72.56 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS ($\alpha = 4$) | 65.80 | 67.68 |
| DEEPSEEK-FINETUNE-8B<br>– TRBS ($\alpha = 5$) | 61.32 | 66.46 |

Table 4: The detailed experimental data of comparative experiments and ablation study for structure anchor protection (SAP) on Python datasets with DeepSeek-R1-Distill-Llama-8B.

ing necessary libraries or packages.

• docstring: A description specifying the functionality of the function, including example inputs and expected outputs.

• prompt: The function declaration along with its docstring.

• canonical_solution: A verified reference solution for the problem.

• test: A test program containing test cases to validate the correctness of solutions.

With 820 problem-solution pairs, HumanEval-X provides a comprehensive benchmark for evaluating multilingual code generation and translation models.

### C.2 DS-1000

DS-1000 is a code generation benchmark consisting of 1,000 data science problems drawn from seven Python libraries like NumPy and Pandas.

| Model | Python | | Java | | JavaScript | | Go | | CPP | |
|---|---|---|---|---|---|---|---|---|---|---|
| | LCS | ACC | LCS | ACC | LCS | ACC | LCS | ACC | LCS | ACC |
| LLAMA3-8B – standard beam search | 24.30 | 57.32 | 56.41 | 51.83 | 23.62 | 52.44 | 30.37 | 43.29 | 19.85 | 42.68 |
| LLAMA3-FINETUNE-8B – standard beam search | 121.14 | 84.15 | 245.37 | 81.10 | 145.80 | 79.27 | 147.87 | 72.56 | 166.59 | 84.76 |
| LLAMA3-FINETUNE-8B – output regeneration | 84.12 | 59.76 | 138.26 | 51.83 | 85.50 | 54.88 | 69.31 | 43.90 | 96.73 | 39.63 |
| LLAMA3-FINETUNE-8B – training data rewrite | 77.94 | 70.12 | 140.57 | 59.76 | 76.15 | 58.54 | 86.86 | 56.71 | 80.73 | 38.41 |
| LLAMA3-FINETUNE-8B – TRBS w\o SAP ($\alpha = 2$) | 96.20 | 78.05 | 189.27 | 73.78 | 114.69 | 72.56 | 93.79 | 59.76 | 123.55 | 70.12 |
| LLAMA3-FINETUNE-8B – TRBS w\o SAP ($\alpha = 3$) | 80.20 | 71.34 | 163.26 | 68.29 | 92.87 | 60.37 | 73.13 | 50.61 | 104.15 | 60.98 |
| LLAMA3-FINETUNE-8B – TRBS w\o SAP ($\alpha = 4$) | 68.39 | 65.85 | 140.88 | 62.80 | 83.31 | 62.20 | 64.20 | 41.46 | 91.97 | 56.71 |
| LLAMA3-FINETUNE-8B – TRBS w\o SAP ($\alpha = 5$) | 58.74 | 64.02 | 126.90 | 56.71 | 76.52 | 60.37 | 55.79 | 39.02 | 81.32 | 53.66 |
| LLAMA3-FINETUNE-8B – TRBS ($\alpha = 2$) | 96.46 | 78.05 | 189.20 | 75.61 | 114.60 | 72.56 | 94.86 | 59.15 | 125.08 | 69.51 |
| LLAMA3-FINETUNE-8B – TRBS ($\alpha = 3$) | 81.79 | 73.78 | 165.48 | 71.34 | 95.95 | 64.02 | 76.65 | 54.27 | 105.34 | 64.02 |
| LLAMA3-FINETUNE-8B – TRBS ($\alpha = 4$) | 73.90 | 70.73 | 147.88 | 68.29 | 88.98 | 65.24 | 67.17 | 50.61 | 95.53 | 62.20 |
| LLAMA3-FINETUNE-8B – TRBS ($\alpha = 5$) | 65.10 | 69.51 | 136.45 | 65.24 | 83.10 | 64.63 | 61.93 | 50.61 | 87.67 | 59.76 |

Table 5: The detailed experimental data of comparative experiments and ablation study for structure anchor protection (SAP) on HumanEval-X.

## D Prompt Details

In this part, we show the prompts used in this study, covering the prompts in finetune, inference, output regeneration, and training data rewrite. The detailed these prompts are shown in Figure 7.

**Finetune/Inference**

```
prompt = {{ system_prompt }}\n\n
### Instruction:\n{{ user_promt }}\n\n
### Response:\n{{ content }}

system_prompt = "Below is an instruction that describes a task.
Write a response that appropriately completes the request.\n\n"

user_prompt = "Complete the code below."
```

**Output Regeneration**

```
prompt = {{ system_prompt }}\n\n
### Instruction:\n{{ user_promt }}\n\n
### Response:\n{{ content }}

system_prompt = "Below is an instruction that describes a task.
Write a response that appropriately completes the request.\n\n"

user_prompt = "Complete the code below. Please do not
respond by simply replicating this:\n{{ example }}"
```

**Training Data Rewrite**

```
prompt = {{ system_prompt }}\n\n
### Instruction:\n{{ user_promt }}\n\n
### Response:\n{{ content }}

system_prompt = "Below is an instruction that describes a task.
Write a response that appropriately completes the request.\n\n"

user_prompt = "Complete the code below by rewriting the
example. Do not copy the example:\n{{ example }}"
```

Figure 7: The prompt templates.