Alignment with Fill-In-the-Middle for Enhancing Code Generation

Houxing Ren[1] **Zimu Lu**[1] **Weikang Shi**[1] **Haotian Hou**[2,5] **Yunqiao Yang**[1]
Ke Wang[1] **Aojun Zhou**[1] **Junting Pan**[1,3] **Mingjie Zhan**[2*] **Hongsheng Li**[1,3,4*]
[1]CUHK MMLab [2]SenseTime Research [3]CPII under InnoHK
[4]Shanghai AI Laboratory [5]Beihang University
renhouxing@gmail.com zhanmingjie@sensetime.com hsli@ee.cuhk.edu.hk

Abstract

The code generation capabilities of Large Language Models (LLMs) have advanced applications like tool invocation and problem-solving. However, improving performance in code-related tasks remains challenging due to limited training data that can be verified with accurate test cases. While Direct Preference Optimization (DPO) has shown promise, existing methods for generating test cases still face limitations. In this paper, we propose a novel approach that splits code snippets into smaller, granular blocks, creating more diverse DPO pairs from the same test cases. Additionally, we introduce the Abstract Syntax Tree (AST) splitting and curriculum training method to enhance the DPO training. Our approach demonstrates significant improvements in code generation tasks, as validated by experiments on benchmark datasets such as HumanEval (+), MBPP (+), APPS, LiveCodeBench, and BigCodeBench. Code and data are available at `https://github.com/SenseLLM/StructureCoder`.

1 Introduction

Large Language Models (LLMs) have significantly enhanced applications like tool invocation and mathematical problem-solving (OpenAI, 2023; Touvron et al., 2023; Jiang et al., 2024; Yang et al., 2024). To improve LLM performance, a common approach involves two stages: supervised fine-tuning (SFT) and alignment. The alignment phase is particularly effective, with Direct Preference Optimization (DPO) (Rafailov et al., 2023) gaining popularity for its simplicity and effectiveness. DPO has been successfully applied in fields like dialogue (Dubey et al., 2024), faithfulness (Bi et al., 2024), and reasoning (Lai et al., 2024).

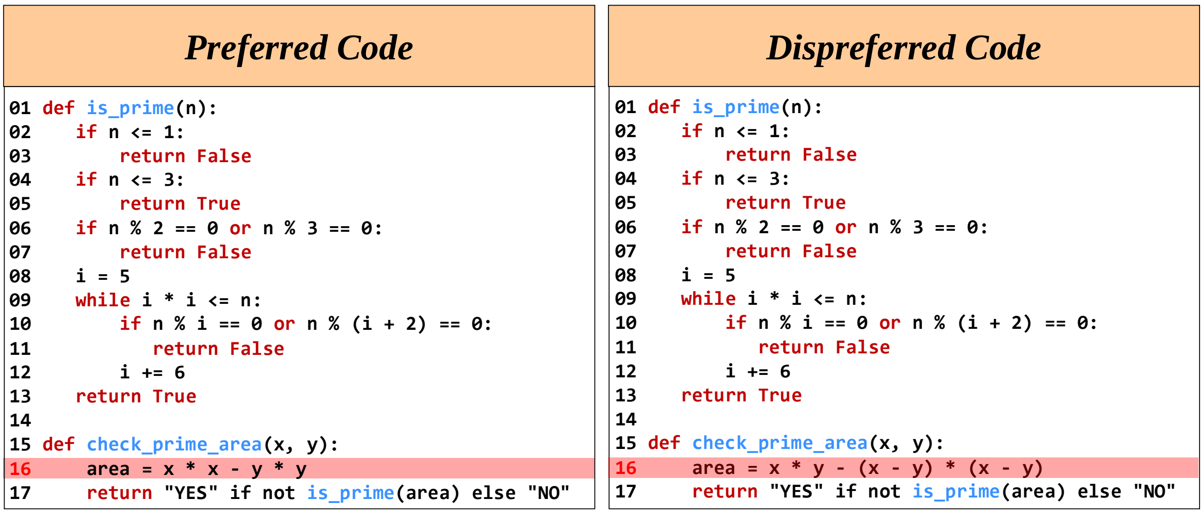
However, despite its success in several domains, DPO has shown limited improvement in code generation tasks, and in some cases, its application may even be counterproductive (Xu et al., 2024). This limitation may stem from the scarcity of training data containing test cases (Zhang et al., 2024a), as seen in training sets like APPS (Hendrycks et al., 2021), which include only 5,000 samples. To address this, PLUM (Zhang et al., 2024a) and CodeDPO (Zhang et al., 2024b) have proposed methods for constructing test cases to expand the training dataset for DPO. However, these approaches still face criticism for their inability to fully guarantee the accuracy of test samples, which limits the achievable performance improvements.

To overcome these challenges, we propose a novel approach that makes more efficient use of the limited, high-quality data available. Inspired by recent advancements in the mathematics domain (Lightman et al., 2024; Lu et al., 2024c,a; Lai et al., 2024), we decompose code snippets into smaller, more granular blocks. This allows for the construction of more detailed and diverse DPO pairs. By treating each block as a target for prediction, the model can fine-tune on smaller code segments, enabling more effective learning and potentially improving performance in code generation tasks. Furthermore, this approach generates a larger number of DPO pairs, making better use of the available training set.

Building on these insights, we introduce a novel method called StructureCoder, which focuses on maximizing the utility of limited training data. Specifically, we leverage the fill-in-the-middle (FIM) approach (Bavarian et al., 2022), an important capability of code LLMs for code completion, to generate fine-grained DPO pairs. By using FIM, we can divide a code snippet into multiple blocks by Abstract Syntax Tree (AST), each serving as a target, and then prompt the model to generate the missing block. Test cases are used to evaluate the correctness of the generated block, thereby constructing accurate DPO pairs. Additionally, we propose a curriculum training method that

[*] Corresponding author.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 8316–8332
November 4-9, 2025 ©2025 Association for Computational Linguistics



Preferred Code

```
01 def is_prime(n):
02    if n <= 1:
03        return False
04    if n <= 3:
05        return True
06    if n % 2 == 0 or n % 3 == 0:
07        return False
08    i = 5
09    while i * i <= n:
10        if n % i == 0 or n % (i + 2) == 0:
11           return False
12        i += 6
13    return True
14
15 def check_prime_area(x, y):
16     area = x * x - y * y
17     return "YES" if not is_prime(area) else "NO"
```

Dispreferred Code

```
01 def is_prime(n):
02    if n <= 1:
03        return False
04    if n <= 3:
05        return True
06    if n % 2 == 0 or n % 3 == 0:
07        return False
08    i = 5
09    while i * i <= n:
10        if n % i == 0 or n % (i + 2) == 0:
11           return False
12        i += 6
13    return True
14
15 def check_prime_area(x, y):
16     area = x * y - (x - y) * (x - y)
17     return "YES" if not is_prime(area) else "NO"
```

Figure 1: A preference pair case in the code generation field. The left is the correct response, and the right is the incorrect response. The only difference between the two responses is in Line 16.

organizes the training set according to the depth of the target block. This method progressively increases training difficulty, leading to better performance.

Our contributions can be summarized as follows:

- We propose the use of FIM to enhance DPO for code LLMs, enabling the effective use of limited training data with test cases.
- We introduce a method for generating accurate FIM data through AST-based block segmentation, and a curriculum training strategy to organize the training set by target block depth, leading to improved performance.
- We conduct extensive experiments on HumanEval (+), MBPP (+), APPS, LiveCodeBench, and BigCodeBench to demonstrate the effectiveness of the proposed method in code-related tasks.

2 Preliminaries

In this section, we introduce the fundamental components of DPO and FIM, followed by an analysis of the limitations of DPO in code generation tasks.

2.1 Direct Preference Optimization

DPO (Rafailov et al., 2023) offers a solution that bypasses reward model training, instead directly fine-tuning the LLM using preference pairs. The DPO loss is defined as

$$\mathcal{L}_{DPO}(y_w, y_l, x) = \\ -\log\sigma\left(\beta\log\frac{\pi_\theta(y_w \mid x)}{\pi_{ref}(y_w \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{ref}(y_l \mid x)}\right),$$

where y_w is the preferred response and y_l is the dispreferred response.

2.2 Fill-In-the-Middle (FIM)

When performing FIM (Bavarian et al., 2022) training, we swap the middle and the suffix segments. Specifically, after splitting, it moves the suffix segment before the middle segment:

$$\text{code} \rightarrow (\text{pre, mid, suf}) \rightarrow (\text{pre, suf, mid}).$$

Then concatenate the three pieces using special tokens as

<PRE> pre <SUF> suf <MID> mid <EOT>.

After that, we use the format to train a causal language model and use the prefix and suffix segments to predict the middle segment in the inference stage.

2.3 Weakness of DPO

Prior work (Rafailov et al., 2024) demonstrates that DPO can implicitly learn token-level reward functions within the Markov Decision Process framework of large language models. This enables DPO-trained models to assign differentiated rewards to individual tokens, effectively identifying those associated with factual inaccuracies while maintaining consistent rewards elsewhere. However, accurately capturing such fine-grained reward signals requires substantial training data, as the quality and scale of the dataset critically influence the learned reward and the performance.

This ability introduces two challenges for DPO in the code field. On the one hand, there are limited training sets containing test cases. On the other hand, the preferred and dispreferred responses often share a similar structure, with only minimal differences in detailed expression. This phenomenon causes DPO to need a larger training data set to learn token-level rewards. This is because when constructing DPO training data, a model is used to generate multiple responses, which are then evaluated against test cases. This method often results in correct responses and incorrect responses being nearly identical, differing only in small details such as a specific expression, an if-block, or a function.

To illustrate the consequences of this scenario, we present an extreme example, as shown in Figure 1. In this case, we assume that only a single segment differs between a correct response y_w and an incorrect response y_l. These sequences are defined as

$$y_w = \{\text{pre}, \text{mid}_w, \text{suf}\},$$
$$y_l = \{\text{pre}, \text{mid}_l, \text{suf}\}.$$

Here, the prefix is the same for both sequences, so the loss for the prefix is zero. The DPO loss can then be expressed as

$$\mathcal{L}(y_w, y_l, x) = \mathcal{L}_{DPO}(\text{mid}_w|\text{suf}, \text{mid}_l|\text{suf}, x|\text{pre}),$$

where $|$ denotes concatenation. Let $x_p = x|\text{pre}$, $x_w = x|\text{pre}|\text{mid}_w$, and $x_l = x|\text{pre}|\text{mid}_l$. The argument of the log sigmoid function in the loss can then be separated into two components:

$$\beta \log \frac{\pi_\theta(\text{mid}_w \mid x_p)}{\pi_{ref}(\text{mid}_w \mid x_p)} - \beta \log \frac{\pi_\theta(\text{mid}_l \mid x_p)}{\pi_{ref}(\text{mid}_l \mid x_p)},$$
$$\beta \log \frac{\pi_\theta(\text{suf} \mid x_w)}{\pi_{ref}(\text{suf} \mid x_w)} - \beta \log \frac{\pi_\theta(\text{suf} \mid x_l)}{\pi_{ref}(\text{suf} \mid x_l)}.$$

The first term corresponds to the loss for the middle segment, which is the key component of the overall loss. The second term, however, has a negative impact, as it implies the model should not generate the suffix based on x_l. However, this is an issue because a prior error is unrelated to the rest segment. For instance, as shown in Figure 1, even though there's an error in Line 16, the remaining function should still be generated like this.

As demonstrated, it is inappropriate to calculate the DPO loss on the suffix, as it leads to negative rewards for correctly generated tokens. Unfortunately, only a small fraction of the generated code typically contains errors; the majority is correct. This implies the need for a large dataset with test cases to make up for the negative impact of the DPO loss on the suffix.

3 Methodology

In Section 2, we analyze the impediments of DPO with limited code training data. In this section, we introduce a novel approach that effectively mitigates these weaknesses. Unlike previous methods (Zhang et al., 2024a,b), which focus on expanding the training dataset, we emphasize optimizing the utilization of the limited available data. We begin by demonstrating how to leverage FIM to avoid the detrimental part of DPO loss. Next, we describe how to construct more accurate FIM data. Finally, we provide an overview of the entire training pipeline.

3.1 FIM Enhanced DPO

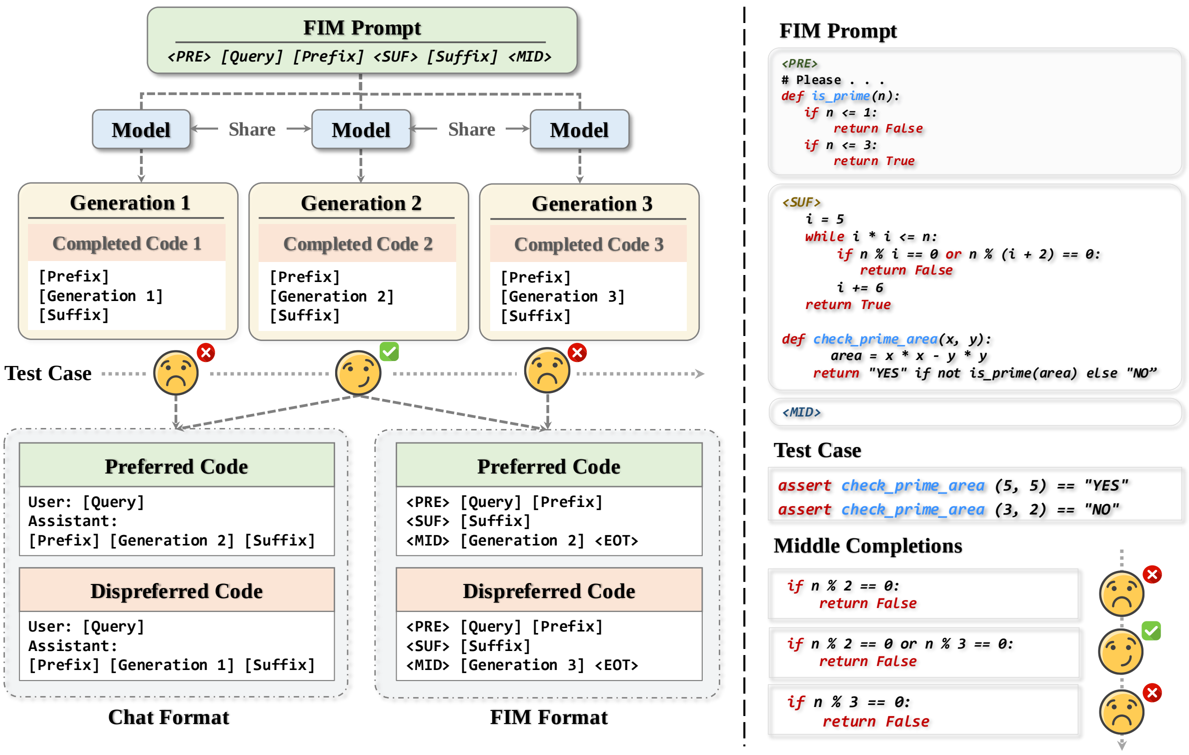
To alleviate the negative impact of the DPO loss on the suffix part, we proposed to combine FIM and DPO to enhance the code generation performance. The FIM ability of code LLMs allows them to generate the middle segment based on the prefix and suffix segments, enabling us to control what is included in the model response flexibly.

As shown in Figure 2, given a code training case, which includes a code problem q, a reference solution c to solve the problem, and several test cases t to test any generation, we first select consequent snippets from the golden code n times to construct the target snippets $M = \{m_1, m_2, \ldots, m_n\}$. This process also generate the corresponding prefixes $P = \{p_1, p_2, \ldots, p_n\}$ and suffixes $S = \{s_1, s_2, \ldots, s_n\}$. Then we construct the FIM prompt with the target snippets, prefixes, and suffixes:

$$\text{<PRE> Convert}(q)\ p_i \text{ <SUF> } s_i \text{ <MID>}, \quad (1)$$

where "Convert" denotes a function converting the text prompts into comments in the code. Then, we task the code LLM to generate m generations for each prefix and suffix pair: $G = \{g_1^{(1)}, \ldots, g_1^{(m)}, \ldots, g_n^{(1)}, \ldots, g_n^{(m)}, \}$ based on the prompt. Finally, we concatenate the generation with the corresponding prefix and suffix, *e.g.,* $p_1 \mid g_1^{(1)} \mid s_1$, and use the test cases t to verify the generation.

For each target snippet, we select a correct response for each incorrect response, based on the



```
<PRE>
# Please . . .
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
```

```
<SUF>
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def check_prime_area(x, y):
        area = x * x - y * y
    return "YES" if not is_prime(area) else "NO”
```

```
<MID>
```

```
assert check_prime_area (5, 5) == "YES"
assert check_prime_area (3, 2) == "NO"
```

```
if n % 2 == 0:
    return False
```

```
if n % 2 == 0 or n % 3 == 0:
    return False
```

```
if n % 3 == 0:
    return False
```

Figure 2: An overview of our FIM-style preference modeling process. A concrete example (right) illustrates the task of completing an is_prime function, with correctness judged via downstream function tests.

minimum edit distance, to construct the preference pair. Then we apply two prompt formats to fine-tune the model. One is the FIM format:

$$\text{<PRE> Convert}(q)\ p_i \text{ <SUF> } s_i \text{ <MID> } g_i^{(j)}. \tag{2}$$

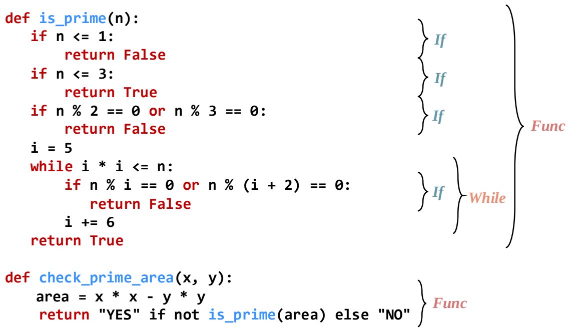
Here, only $g_i^{(j)}$ is regarded as the response and included in the DPO loss. Another format is the Chat format:

$$\text{User: } q. \text{ Assistant: } p_i\ g_i^{(j)}\ s_i. \tag{3}$$

Also, only $g_i^{(j)}$ is calculated in the DPO loss. This format is designed to preserve the model's chat capabilities. During training, we randomly choose one of two formats for each sample by drawing from a Bernoulli distribution with probability α, which determines the proportion of samples using each format. This strategy enables the model to be fine-tuned on shorter code snippets, thereby improving learning efficiency.

3.2 AST Enhanced FIM

As discussed in Section 2, errors in the middle segment are independent of those in the suffix segment. Therefore, it is essential to select appropriate consequent code snippets from the golden label.


```
def is_prime(n):
   if n <= 1:
       return False
   if n <= 3:
       return True
   if n % 2 == 0 or n % 3 == 0:
       return False
   i = 5
   while i * i <= n:
       if n % i == 0 or n % (i + 2) == 0:
          return False
       i += 6
   return True

def check_prime_area(x, y):
    area = x * x - y * y
    return "YES" if not is_prime(area) else "NO"
```

Figure 3: Illustration of our AST-based segmentation strategy. This segmentation ensures each middle segment is syntactically and semantically coherent, enabling more effective FIM-based fine-tuning.

The ideal snippets should have the following characteristics to ensure effective model learning and improved performance:

- The selected code snippets should be distinct blocks to maintain clear boundaries between different functional units, ensuring that each segment functions independently without relying on adjacent ones.
- The selected code snippets should encompass a variety of code structures to enable the model to learn a broad range of coding patterns.

Inspired by previous work (Gong et al., 2024), we parse a code into an Abstract Syntax Tree (AST). As illustrated in Figure 3, we first parse the golden label into an AST, then select several blocks as the target code snippets. In this approach, we consider only four node types in the AST: *i.e.,* if block, for block, while block, and function block.

By analyzing the AST, we can extract syntactically complete and functionally independent code blocks, which reduces the risk of including partial or entangled code, ensures that the middle segments are self-contained and coherent, and exposes the model to a broader range of distinct structural patterns. For instance, as shown in Figure 3, the first three if blocks handle distinct conditions and thus operate independently, focusing solely on their respective logic and ignoring others. Additionally, the four selected node types represent the core building blocks of Python code, capturing a diverse range of structures commonly encountered in Python programming.

3.3 Training and Discussion

To effectively leverage all the blocks selected from AST, we propose a curriculum learning strategy (Bengio et al., 2009) that orders the training data based on the length of code snippets. The goal is to enable the model to first focus on mastering token-level rewards for shorter, simpler code snippets before gradually progressing to longer, more complex ones, and finally the whole codes. This strategy leverages the inherent structure of code, where shorter snippets tend to contain simpler constructs and fewer dependencies, making them easier to learn at the token level.

Specifically, we first sort the training samples by the number of lines in the corresponding code snippets. During the initial training stages, the model is fine-tuned on shorter code snippets, which typically have fewer tokens and simpler logic. This allows the model to learn the fundamental code patterns without being overwhelmed by the complexity of longer code blocks. As the training procedure, we progressively introduce longer snippets, which feature more intricate logic, dependencies, and structures.

In summary, the training begins by selecting target code snippets by AST segmentation from the golden code, using FIM to generate middle segments based on the prefix and suffix. Then the training data is ordered by code snippet length, with the model first learning from shorter, simpler code and progressively moving to longer, more complex snippets, ensuring a structured and efficient learning process. Finally, we fine-tune the code LLMs with the sorted training data and randomly select the prompt format (FIM format or chat format) by drawing from a Bernoulli distribution with probability α in each training step.

4 Experiments

In this section, we construct extensive experiments to demonstrate the effectiveness of the proposed method and analyze the proposed method.

4.1 Experimental Setup

Training Dataset. Our training dataset is the APPS training set (Hendrycks et al., 2021), which is collected from different open-access coding websites such as Codeforces, Kattis, and more. The training set includes 5,000 samples and each sample contains a query, several golden labels, and several test cases.

Test Dataset. We evaluate our method on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), two of the most widely used benchmarks for code generation. Considering the insufficiency of test cases in these benchmarks, Liu et al. (2023) proposed HumanEval+ and MBPP+, which contain 80×/35× more tests. We also evaluate our method on APPS (Hendrycks et al., 2021) and LiveCodeBench (Jain et al., 2024), two datasets with more difficult code problems. In addition, we evaluate our method on BigCodeBench (Zhuo et al., 2024), which challenged LLMs to invoke multiple function calls as tools.

Implementation Details. We test our methods on Qwen2.5-Coder-Instruct 1.5B/3B/7B (Hui et al., 2024). We first use Black[1] to format all golden labels in the training set, and then use AST[2] to parse the formatted code and extract all desired blocks, *i.e.,* if block, for block, while block, function block. Then we use the code LLMs to generate 5 responses based on the FIM prompt, with top_p = 0.95 and temperature = 0.7. After that, we use the test cases to evaluate the generated responses and select one preferred response and one dispreferred response (if there is no correct generation or no incorrect generation, we will discard this case). For training, we fine-tune all models for 3 epochs.

[1] https://github.com/psf/black

[2] https://docs.python.org/3/library/ast.html

Method	HumanEval		MBPP		APPS	LiveCodeBench	BigCodeBench	
	Base	Plus	Base	Plus			Full	Hard
				Closed-Source Models				
Claude-3.5-Sonnet	92.1	86.0	91.0	74.6	-	37.1	45.3	23.6
GPT-4o	92.1	86.0	86.8	72.5	-	33.0	50.1	25.0
o1-mini	97.6	90.2	93.9	78.3	-	54.1	46.3	23.0
o1-preview	95.1	88.4	93.4	77.8	-	42.5	49.3	27.7
				Open-Source Models				
DS-Coder-1.3B	65.9	60.4	65.3	54.8	$\underline{5.7}$	$\underline{4.1}$	22.8	3.4
Qwen2.5-Coder-1.5B	70.7	66.5	69.2	59.4	$\underline{7.7}$	$\underline{6.7}$	$\underline{25.2}$	$\underline{4.1}$
w/ DPO	74.4	69.5	73.5	63.8	9.3	7.8	28.6	6.8
w/ KTO	74.4	70.1	74.1	62.4	9.4	9.4	28.0	7.4
w/ Focused-DPO	72.6	68.3	72.8	62.7	9.2	9.1	**29.2**	7.4
w/ StructureCoder	**75.6**‡	**71.3**‡	**75.7***†‡	**64.8***†‡	**9.9***†‡	**10.8***‡	**29.2***†	**8.1***‡
Qwen2.5-Coder-3B	84.1	80.5	73.6	62.4	$\underline{10.9}$	$\underline{10.4}$	35.8	14.2
w/ DPO	84.8	80.5	75.4	64.0	12.7	10.8	37.4	12.8
w/ KTO	81.1	77.4	73.3	63.0	12.4	10.8	36.6	11.5
w/ Focused-DPO	**86.0**	81.1	**75.7**	64.3	**13.9**	10.8	37.9	14.5
w/ StructureCoder	**86.0***†	**82.3***†	**75.7***†	**64.6***†‡	**13.9***†	**13.4***†‡	**38.4***†‡	**16.9***†‡
CodeLlama-7B	40.9	33.5	54.0	44.4	$\underline{4.3}$	$\underline{7.1}$	21.9	3.4
DS-Coder-6.7B	74.4	71.3	74.9	65.6	$\underline{8.5}$	$\underline{10.4}$	35.5	10.1
Qwen2.5-Coder-7B	88.4	84.1	83.5	71.7	$\underline{16.2}$	$\underline{16.8}$	41.0	18.2
w/ DPO	88.4	84.1	83.1	71.4	19.8	17.5	**41.8**	19.1
w/ KTO	89.0	84.1	85.4	72.5	19.0	16.9	41.0	18.2
w/ Focused-DPO	89.0	84.8	85.4	72.5	20.0	17.9	41.1	18.2
w/ StructureCoder	**90.9***†‡	**87.2***†‡	**85.7***	**74.3***†	**20.1***†	**18.3***†‡	**41.8**†	**19.6***†‡

Table 1: Pass@1 accuracy on HumanEval (+), MBPP (+), APPS, LiveCodeBench (July 2024 - January 2025), and BigCodeBench. The best results of each base are in bold, and results unavailable are left blank. The results re-evaluated on our end are marked with an underline. $*$, †, and ‡ denote that our method significantly outperforms DPO, KTO, and Focused-DPO at the level of 0.05, respectively. Note that all results of the LiveCodeBench have been re-evaluated due to the dataset update. We also report the re-evaluated results of the BigCodeBench of Qwen2.5-Coder-1.5B-Inst because the reproduced results differed significantly from the original paper.

We employ an RMSProp optimizer (Graves, 2013) with a learning rate of 1e-6, a 0.05 warm-up ratio, and a cosine scheduler. We set the batch size as 128 and the max sequence length as 2048. We set the hyperparameter $\alpha = 0.5$. To efficiently train the computationally intensive models, we simultaneously employ VLLM (Kwon et al., 2023), DeepSpeed (Rajbhandari et al., 2020) and Flash Attention (Dao, 2023). On 8 NVIDIA A800 80GB GPUs, the experiments on 1.5B, 3B, and 7B models take 8 hours, 10 hours, and 16 hours, respectively.

4.2 Evaluation

Baseline. We employ DPO (Rafailov et al., 2023), KTO (Ethayarajh et al., 2024), and Focused-DPO (Zhang et al., 2025) as our baseline. For the DPO and KTO, we use the chat format to generate 5 responses for each problem. For Focused-DPO, we follow the paper to generate 10 responses and select the 5000 samples with the longest common prefix and suffix. The other pipeline and hyperparameters are the same as the implementation details of StructureCoder.

Results. Table 1 shows the pass@1 accuracy of different models on HumanEval, MBPP, APPS, LiveCodeBench, and BigCodeBench. Based on the results, we have the following findings:

(1) The proposed StructureCoder consistently outperforms standard DPO and KTO across all

Method	HumanEval		MBPP		APPS	LiveCodeBench	BigCodeBench	
	Base	Plus	Base	Plus			Full	Hard
StructureCoder	75.6	71.3	**75.7**	**64.8**	**9.9**	**10.8**	**29.2**	**8.1**
w/ DPO	74.4	69.5	73.5	63.8	9.3	7.8	28.6	6.8
w/ DPO (Data Equal)	66.5	59.8	71.4	59.0	2.0	4.1	25.9	6.1
w/o AST	75.0	69.5	74.1	62.4	9.9	9.7	28.8	3.4
w/o Curriculum	**78.7**	**73.2**	73.8	64.3	9.8	9.0	28.3	7.4
w/ $\alpha = 0$	75.0	70.7	74.1	62.4	9.6	8.6	28.7	8.1
w/ $\alpha = 1$	78.7	72.0	74.1	64.0	9.6	9.7	29.1	8.1
w/ suf loss	77.4	70.7	75.4	64.6	9.8	8.2	29.0	7.4
w/ pre & suf loss	76.2	70.7	74.9	64.6	9.7	8.1	29.1	7.4
w/o suf	73.2	69.5	75.1	62.4	9.8	9.7	29.1	**8.1**

Table 2: Ablation results based on Qwen2.5-Coder-1.5B-Instruct. The score is Pass@1 accuracy.

models on a variety of code-related tasks. Specifically, for Qwen2.5-Coder-1.5B-Instruct, StructureCoder surpasses DPO by an average of 1.5 points across all test sets. Similarly, for Qwen2.5-Coder-3B-Instruct and Qwen2.5-Coder-7B-Instruct, StructureCoder outperforms DPO by an average of 1.6 points across all test sets. The similar increase in different sizes demonstrates the effectiveness of the proposed method.

(2) The proposed StructureCoder shows a more significant improvement over DPO on most test sets compared to the APPS test set. For Qwen2.5-Coder-1.5B-Instruct, StructureCoder exceeds DPO by an average of 1.5 points on all test sets except for the APPS test set, where the improvement is only 0.6 points. This suggests that the proposed method enables the model to learn more diverse code patterns and achieve better generalization.

(3) The proposed StructureCoder demonstrates comparable performance to Focused-DPO on Qwen2.5-Coder-7B-Instruct, but achieves a significant improvement on Qwen2.5-Coder-1.5B-Instruct. This is because generations from the weaker model tend to have shorter comment prefixes and suffixes, which diminishes the effectiveness of Focused-DPO. In contrast, our method does not rely on aligning specific features of the generations, leading to consistent improvements across all three models.

4.3 Detailed Analysis

4.3.1 Ablation Study

Here, we check how each component contributes to the final performance. We prepare two group variants of our method based on Qwen2.5-Coder-1.5B-Instruct: (1) The first group is related to DPO. w/ DPO denotes constructing training data w/o FIM. w/ DPO (Data Equal) denotes training with more epochs to ensure the total training samples are the same as StructureCoder. (2) The second group is related to the proposed components. w/ AST denotes selecting target code snippets randomly. w/ Curriculum denotes training with random order. w/ $\alpha = 0$ denotes training only with the standard format. w/ $\alpha = 1$ denotes training only with the FIM format.

Table 2 presents the pass@1 accuracy for various model variants. Based on the results, we have the following findings:

(1) As shown, all variants perform worse than StructureCoder across most test sets, except for HumanEval. These results suggest that all components (*i.e.,* AST enhanced FIM, curriculum learning, and mixing FIM training format and Chat format) are critical for enhancing performance.

(2) Furthermore, w/ DPO (Data Equal) performs worse than w/ DPO, indicating that DPO benefits more from diverse, large-scale training data than from simply increasing the number of epochs. Increasing epochs can lead to overfitting. In contrast, StructureCoder effectively generates more diverse training data through FIM, resulting in improved performance.

(3) Lastly, w/ Curriculum and w/ $\alpha = 1$ outperform StructureCoder on HumanEval. A potential explanation for this is that the HumanEval test cases differ significantly from those in the APPS, making fine-tuning with longer code during the

final stage detrimental to HumanEval performance.

4.3.2 Effect of Loss on Prefix and Suffix

As we analyzed in Section 2, the DPO loss on the prefix segment does not impact the optimization, but the DPO loss on the suffix segment harms the optimization. Here, we conduct experiments to check the effect of loss on the prefix segment and the suffix segment. Specifically, we prepare three variants based on Qwen2.5-Coder-1.5B-Instruct. w/ suf loss computes the loss on both the middle and the suffix segments. w/ pre & suf loss computes the loss on all segments. w/o suf remove suffix in the whole pipeline, *a.k.a.*, we only use the query and the prefix to prompt the model to generate various generations, effectively setting the suffix to be empty.

Table 2 presents the pass@1 accuracy for the variants. As shown in the table, w/ suf loss significantly performs worse than StructureCoder on LiveCodeBench, which is the most difficult test set. This indicates that the DPO loss on the suffix segment hurts the optimization. However, w/ suf loss and w/ pre & suf loss perform similarly. This indicates that the DPO loss on the prefix segment does not impact the optimization. This phenomenon is less noticeable on other datasets because they are simpler. The difficulty of LiveCodeBench highlights the impact of the DPO loss on the suffix segment, while simpler datasets don't show the same effect. Finally, w/o suf performs worse than StructureCoder in most settings. This is because concatenating the middle and suffix spans results in overly long input sequences, which may hinder the model's ability to focus on critical code blocks. This observation aligns with our theoretical insights discussed in Section 2.3.

5 Related Work

5.1 Large Language Models for Code

Previous works have introduced LLMs for the code domain. OpenAI introduced Codex (Chen et al., 2021), and Google introduced PaLM-Coder (Chowdhery et al., 2023). There are also several open-source LLMs for the code domain, such as CodeGen (Nijkamp et al., 2023), Incoder (Fried et al., 2023), SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023), StarCoder-2 (Lozhkov et al., 2024), CodeGeeX (Zheng et al., 2023), Code Llama (Rozière et al., 2023), and DeepSeek-Coder (Guo et al., 2024). Recently, Deepseek-Coder-V2 (DeepSeek-AI et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) have been proposed, which achieve performance close to that of closed-source models.

Instruction tuning is demonstrated to be an effective method to improve the ability of LLMs in specific domains. Code Alpaca (Chaudhary, 2023) applied Self-Instruct (Wei et al., 2022) to fine-tune LLMs with ChatGPT-generated instructions. WizardCoder (Luo et al., 2023) proposed Code Evol-Instruct, which evolves Code Alpaca data using the ChatGPT to generate more complex and diverse datasets. MagiCoder (Wei et al., 2023), WaveCoder (Yu et al., 2023), and InverseCoder (Wu et al., 2024) proposed some methods to make full use of source code. OpenCodeInterpreter (Zheng et al., 2024), AutoCoder (Lei et al., 2024) and ReflectionCoder (Ren et al., 2024b) proposed to utilize a compiler to enhance code LLMs.

5.2 Alignment for Code

Reinforcement learning-based alignment methods have been shown to improve LLM performance. DPO (Rafailov et al., 2023), though widely adopted, provides limited gains in code generation tasks (Xu et al., 2024). PLUM (Zhang et al., 2024a) uses GPT-4 to generate test cases for code validation and ranking. CodeDPO (Zhang et al., 2024b) applies a PageRank-inspired algorithm for iterative preference scoring. StepCoder (Dou et al., 2024) employs PPO (Schulman et al., 2017) to optimize general code generation, and CodeOptimise (Gee et al., 2024) focuses on improving runtime efficiency. Concurrent with our work, Focused-DPO (Zhang et al., 2025) also identified the issue of common prefixes and suffixes. However, their approach heavily depends on model generations that contain these patterns, whereas our method eliminates such dependency.

Unlike previous DPO-based approaches to code generation, our approach focuses on maximizing the utility of limited data to construct more effective preference pairs. While existing methods often rely on larger datasets or complex augmentation strategies, we take a data-efficient approach by making the most of even small-scale datasets. By introducing FIM (Bavarian et al., 2022), we can efficiently extract high-quality preference pairs, significantly improving the alignment of code generation models while utilizing a limited amount of labeled data.

6 Conclusion

In this paper, we introduced a novel approach to enhance code generation performance for LLMs using the fill-in-the-middle (FIM). By generating fine-grained DPO pairs from limited test cases and employing a curriculum training method, we maximized the utility of available data, leading to improved model performance. Our experiments on benchmarks like HumanEval (+), MBPP (+), APPS, LiveCodeBench, and BigCodeBench demonstrated the effectiveness of our approach in code-related tasks. Future work could explore further refinements, such as alternative segmentation strategies or additional data sources.

Limitations

One key limitation of our approach is that it requires the model to possess strong Fill-In-the-Middle (FIM) capabilities, which are essential for effectively generating fine-grained DPO pairs. However, most current models, after supervised fine-tuning, may struggle to maintain high FIM performance. As a result, the effectiveness of our method is closely tied to the underlying model's proficiency in FIM, a capability that is not yet universally available in existing LLMs. This reliance on FIM limits the applicability of our approach to models that have been specifically optimized for this task. Another limitation is that our method is currently focused solely on the code generation domain. While we believe that similar phenomena may exist in other domains, our approach has not yet been tested outside the code generation field. Further research would be required to explore the transferability to other closed-question tasks.

Ethics Statement

The model utilized in this paper, Qwen2.5-Coder (Li et al., 2023), is licensed for academic research purposes[3]. Furthermore, the data employed in this study, APPS (Hendrycks et al., 2021), is also licensed for academic research purposes[4].

Acknowledgment

This project is funded in part by National Key R&D Program of China Project 2022ZD0161100, by the Centre for Perceptual and Interactive Intelligence (CPII) Ltd under the Innovation and Technology Commission (ITC)'s InnoHK, and in part by NSFC-RGC Project N_CUHK498/24.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM.

Baolong Bi, Shaohan Huang, Yiwei Wang, Tianchi Yang, Zihan Zhang, Haizhen Huang, Lingrui Mei, Junfeng Fang, Zehao Li, Furu Wei, Weiwei Deng, Feng Sun, Qi Zhang, and Shenghua Liu. 2024. Context-dpo: Aligning language models for context-faithfulness. *CoRR*, abs/2412.15280.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie

[3] https://github.com/QwenLM/Qwen2.5-Coder

[4] https://github.com/hendrycks/apps

Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, abs/2307.08691.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, and S. S. Li. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *CoRR*, abs/2402.01391.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. KTO: model alignment as prospect theoretic optimization. *CoRR*, abs/2402.01306.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2024. Code-optimise: Self-generated preference data for correctness and efficiency. *CoRR*, abs/2406.12502.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: structure-aware pretraining for code generation and understanding. *CoRR*, abs/2401.03003.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

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

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *CoRR*, abs/2401.04088.

Dongzhi Jiang, Ziyu Guo, Renrui Zhang, Zhuofan Zong, Hao Li, Le Zhuo, Shilin Yan, Pheng-Ann Heng, and Hongsheng Li. 2025. T2i-r1: Reinforcing image generation with collaborative semantic-level and token-level cot. *arXiv preprint arXiv:2505.00703*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM.

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. 2024. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *CoRR*, abs/2406.18629.

Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. Autocoder: Enhancing code large language model with aiev-instruct. *CoRR*, abs/2405.14906.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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 *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch.

Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024a. Mathgenie: Generating synthetic data with question back-translation for enhancing mathematical reasoning of llms.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024b. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024c. Step-controlled DPO: leveraging stepwise error for enhanced mathematical reasoning. *CoRR*, abs/2407.00782.

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

Bingqi Ma, Zhuofan Zong, Guanglu Song, Hongsheng Li, and Yu Liu. 2024. Exploring the role of large language models in prompt encoding for diffusion models. *arXiv preprint arXiv:2406.11831*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Rafael Rafailov, Joey Hejna, Ryan Park, and Chelsea Finn. 2024. From *r* to q^*: Your language model is secretly a q-function. *CoRR*, abs/2404.12358.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 20. IEEE/ACM.

Houxing Ren, Linjun Shou, Jian Pei, Ning Wu, Ming Gong, and Daxin Jiang. 2022a. Lexicon-enhanced self-supervised training for multilingual dense retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 444–459. Association for Computational Linguistics.

Houxing Ren, Linjun Shou, Ning Wu, Ming Gong, and Daxin Jiang. 2022b. Empowering dual-encoder with query generator for cross-lingual dense retrieval. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 3107–3121. Association for Computational Linguistics.

Houxing Ren, Mingjie Zhan, Zhongyuan Wu, and Hongsheng Li. 2024a. Empowering character-level text infilling by eliminating sub-tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 3253–3267. Association for Computational Linguistics.

Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. 2024b. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. *CoRR*, abs/2405.17057.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Ke Wang, Junting Pan, Weikang Shi, Zimu Lu, Mingjie Zhan, and Hongsheng Li. 2024. Measuring multimodal mathematical reasoning with math-vision dataset.

Ke Wang, Junting Pan, Linda Wei, Aojun Zhou, Weikang Shi, Zimu Lu, Han Xiao, Yunqiao Yang, Houxing Ren, Mingjie Zhan, and Hongsheng Li. 2025. Mathcoder-vl: Bridging vision and code for enhanced multimodal mathematical reasoning.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *CoRR*, abs/2312.02120.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct. *CoRR*, abs/2407.05700.

Shusheng Xu, Wei Fu, Jiaxuan Gao, Wenjie Ye, Weilin Liu, Zhiyu Mei, Guangju Wang, Chao Yu, and Yi Wu. 2024. Is DPO superior to PPO for LLM alignment? A comprehensive study. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024. Qwen2.5 technical report. *CoRR*, abs/2412.15115.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *CoRR*, abs/2312.14187.

Dylan Zhang, Shizhe Diao, Xueyan Zou, and Hao Peng. 2024a. Plum: Improving code lms with execution-guided on-policy preference learning driven by synthetic test cases. *CoRR*, abs/2406.06887.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024b. Codedpo: Aligning code models with self generated and verified source code. *CoRR*, abs/2410.05605.

Kechi Zhang, Ge Li, Jia Li, Yihong Dong, Jia Li, and Zhi Jin. 2025. Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points. *CoRR*, abs/2502.11475.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. 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, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 5673–5684. ACM.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *CoRR*, abs/2402.14658.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

Zhuofan Zong, Dongzhi Jiang, Bingqi Ma, Guanglu Song, Hao Shao, Dazhong Shen, Yu Liu, and Hongsheng Li. 2024a. Easyref: Omni-generalized group image reference for diffusion models via multimodal llm. *arXiv preprint arXiv:2412.09618*.

Zhuofan Zong, Bingqi Ma, Dazhong Shen, Guanglu Song, Hao Shao, Dongzhi Jiang, Hongsheng Li, and Yu Liu. 2024b. Mova: Adapting mixture of vision experts to multimodal context. *arXiv preprint arXiv:2404.13046*.

Appendix

A Dataset Construction

To evaluate the structural diversity of training data constructed for our method, we performed a statistical analysis of code blocks extracted via Abstract Syntax Tree (AST) parsing. Specifically, we retained four categories of syntactic nodes as middle targets for the FIM task: if, for, while, and def (function definitions). The figure below presents the distribution of these node types in the final training dataset.

As shown in Figure 4, among all segments used in FIM training, if blocks constitute the largest proportion (41.54%), followed by for loops (32.09%), def (function) blocks (18.63%), and while loops (7.74%). This distribution indicates that the selected code snippets exhibit substantial structural variety, with conditional (if) and iterative (for and while) constructs dominating the sample pool. Such a mix helps the model learn frequent control flow patterns more effectively in FIM tasks. Although while and def blocks appear less frequently, they still contribute essential control and functional semantics, ensuring the structural completeness of the training examples.

To construct the training dataset, we follow a systematic pipeline that parses source code into abstract syntax trees, generates fill-in-the-middle (FIM) prompts, evaluates model completions against test cases, and derives preference pairs from completions. The detailed procedure is outlined in Algorithm 1.

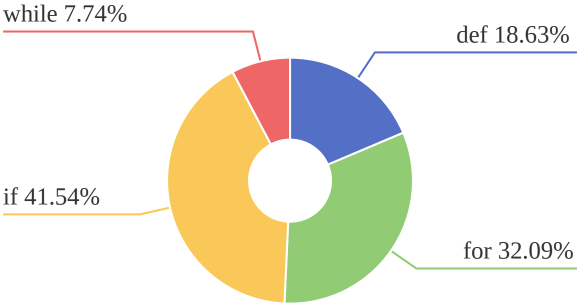


Figure 4: Distribution of extracted code blocks based on AST node types. This reflects the syntactic diversity in the training corpus and validates the representativeness of selected blocks.

B Insights of StructureCoder

B.1 Benefits of FIM

Fill-in-the-Middle (FIM) (Bavarian et al., 2022) has been shown to preserve the model's ability to generate code effectively (Li et al., 2023; Rozière et al., 2023; Ren et al., 2024a), while offering several distinct advantages that enhance overall model performance and robustness. By requiring generation conditioned on both preceding and succeeding context, FIM strengthens contextual understanding and promotes deeper semantic reasoning. Rather than replacing standard code generation, FIM acts as a powerful complement—particularly well-suited for practical scenarios such as code refactoring, patching, and bug fixing. Its bidirectional formulation improves generalization and equips models to handle complex code dependencies more effectively. Our empirical results further demonstrate that FIM-based training leads to notable gains in code generation quality, underscoring its value as a targeted and practical enhancement to existing methodologies.

B.2 Comparison with Step-DPO

While our proposed method shares some similarities with Step-DPO regarding loss construction and data generation, it offers distinct improvements in the following aspects:

(1) Step-DPO relies on labeling intermediate reasoning steps either through human annotation, which is prohibitively expensive, or through automated labeling methods such as GPT-4, which introduces potential biases as highlighted by recent studies (DeepSeek-AI et al., 2025). In contrast, our proposed method utilizes FIM to automatically generate and assess intermediate reasoning fragments based on objective evaluation criteria from test sam-

ples. By eliminating the reliance on costly human annotation and avoiding the biases associated with automated labeling tools, our method offers an unbiased alternative for training and evaluating reasoning capabilities in large language models.

(2) Annotating individual steps in Step-DPO can be inherently ambiguous, especially for long Chain-of-Thought (CoT). An incorrect intermediate step can sometimes contribute meaningfully to reaching a correct final answer — for example, models like o1 often make a mistake in early reasoning steps, such as miscalculating a quantity or misunderstanding part of the problem, but then later detect the inconsistency in their own logic and correct for it in the final answer. In such cases, although the intermediate step is technically incorrect, it still plays a functional role in producing the right outcome. This makes it difficult to label individual steps as preferred or dispreferred definitively. Our approach overcomes this issue by fixing the suffix of the code, requiring the generated intermediate code (the middle segment) to bridge the given prefix and suffix coherently. This design fundamentally eliminates ambiguity in annotation, clearly delineating preferred and dispreferred middle segments.

C Additional Experiment

C.1 FIM Evaluaion

To thoroughly assess the Fill-In-the-Middle (FIM) capabilities of our proposed method, we conducted a series of targeted evaluations using the APPS benchmark. Specifically, we curated a set of 22,044 samples by randomly masking contiguous code spans from the reference solutions. The models were then tasked with accurately reconstructing these masked segments solely based on the surrounding context, thereby simulating realistic code completion scenarios.

As summarized in Table 3, the application of our method consistently improves FIM performance across all evaluated model scales. Notably, our approach achieves the highest Pass@1 accuracy in each model category, surpassing several strong baselines. These findings underscore the robustness and generalizability of our method in enhancing middle-span code generation, demonstrating its potential as a principled strategy for strengthening contextual reasoning in large language models for programming tasks.

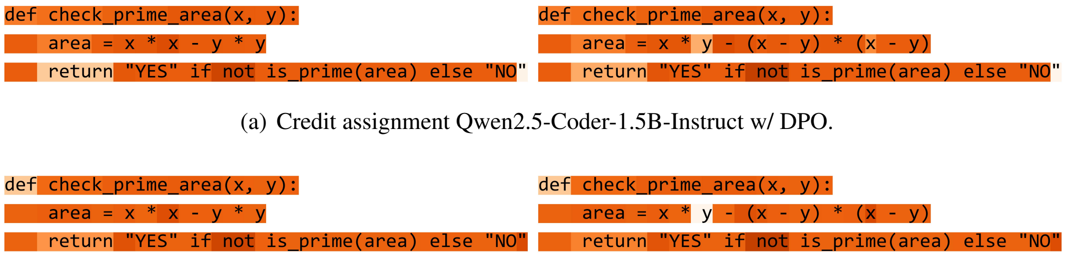
Model	FIM Pass@1
Qwen-2.5-Coder-1.5B-Inst	27.1
w/ DPO	27.5
w/ KTO	27.4
w/ Focused-DPO	27.4
w/ StructureCoder	**28.6**
Qwen-2.5-Coder-3B-Inst	31.1
w/ DPO	31.5
w/ KTO	31.6
w/ Focused-DPO	31.7
w/ StructureCoder	**32.9**
Qwen-2.5-Coder-7B-Inst	35.7
w/ DPO	35.6
w/ KTO	35.9
w/ Focused-DPO	36.1
w/ StructureCoder	**36.7**

Table 3: Pass@1 accuracy on the FIM task.

C.2 Case Study

Here, we use a case from Qwen2.5-Coder-1.5B-Instruct, Qwen2.5-Coder-3B-Instruct and Qwen2.5-Coder-7B-Instruct to show the effectiveness of the proposed method. Specifically, we compute the DPO reward ($r(s, a) = \beta \log \pi_\theta(s|a) - \beta \log \pi_{ref}(s|a)$) for each token in both correct and incorrect responses. The DPO is used as a baseline for comparison.

As shown in Figure 5, the error is introduced in Line 2, and the proposed method, StructureCoder, successfully identifies the token corresponding to the erroneous statement ($x * x$ and $x * y$). For Qwen2.5-Coder-3B-Instruct, as shown in Figure 6, the proposed StructureCoder successfully assigns high reward to the token corresponding to the erroneous statement ($len(bits) - 1$ and $len(bits)$). For Qwen2.5-Coder-7B-Instruct, as shown in Figure 7, the proposed StructureCoder effectively highlights he erroneous statement ($n + i$ and $n - i$). These demonstrate the ability of our method to localize specific errors and focus attention on the relevant details. On the other hand, the DPO fails to detect these errors, highlighting its limitation in handling errors with high precision, especially when the training data is sparse. While the DPO might perform well with abundant training examples, its performance drops in the presence of limited data, making it less effective for tasks requiring fine-grained error detection in these conditions.



```python
def check_prime_area(x, y):
    area = x * x - y * y
    return "YES" if not is_prime(area) else "NO"
```

```python
def check_prime_area(x, y):
    area = x * y - (x - y) * (x - y)
    return "YES" if not is_prime(area) else "NO"
```

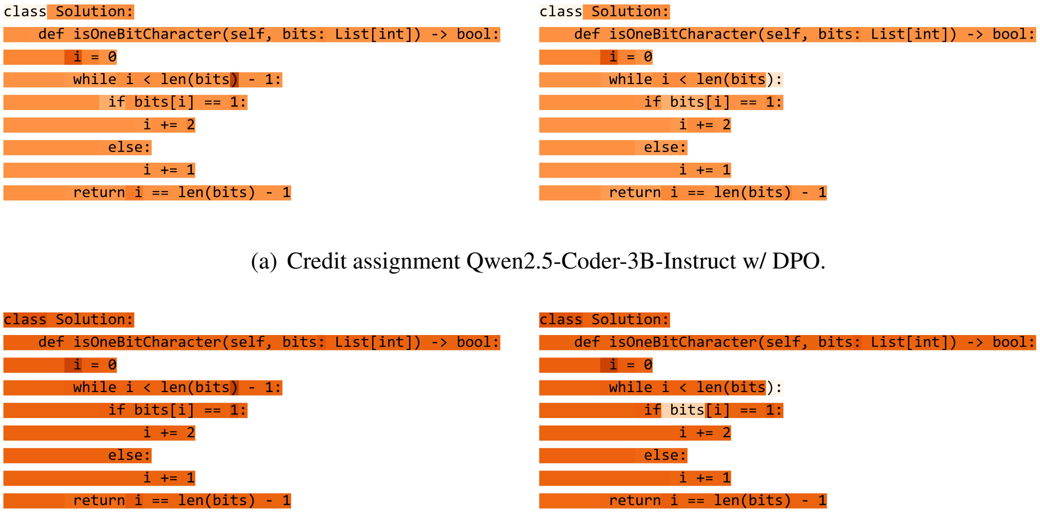
(a) Credit assignment Qwen2.5-Coder-1.5B-Instruct w/ DPO.

```python
def check_prime_area(x, y):
    area = x * x - y * y
    return "YES" if not is_prime(area) else "NO"
```

```python
def check_prime_area(x, y):
    area = x * y - (x - y) * (x - y)
    return "YES" if not is_prime(area) else "NO"
```

(b) Credit assignment Qwen2.5-Coder-1.5B-Instruct w/ StructureCoder.

Figure 5: Credit assignment with different methods. Due to the limited space, we omit the previous function, only keeping the key function. The left is the correct response and the right is the incorrect response (error is introduced in Line 2). Each token is colored corresponding to the DPO implicit reward (darker is higher).



```python
class Solution:
    def isOneBitCharacter(self, bits: List[int]) -> bool:
        i = 0
        while i < len(bits) - 1:
            if bits[i] == 1:
                i += 2
            else:
                i += 1
        return i == len(bits) - 1
```

```python
class Solution:
    def isOneBitCharacter(self, bits: List[int]) -> bool:
        i = 0
        while i < len(bits):
            if bits[i] == 1:
                i += 2
            else:
                i += 1
        return i == len(bits) - 1
```

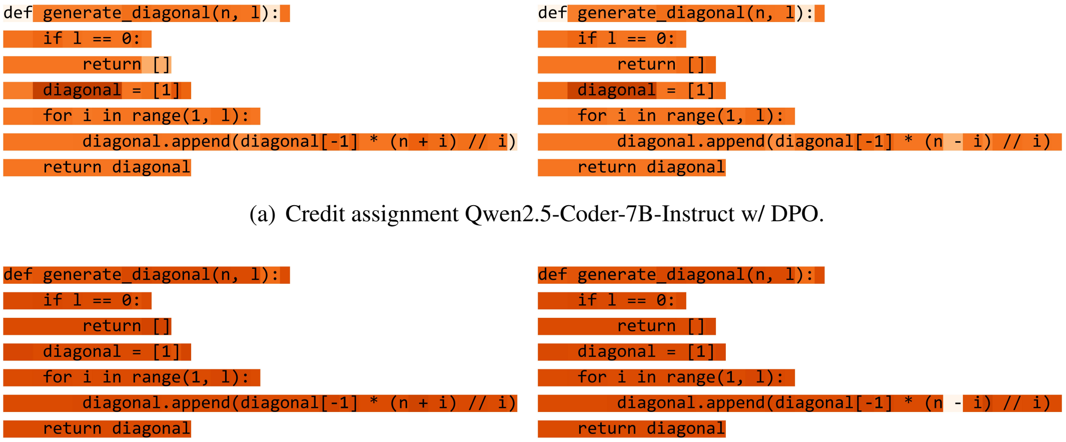
(a) Credit assignment Qwen2.5-Coder-3B-Instruct w/ DPO.

```python
class Solution:
    def isOneBitCharacter(self, bits: List[int]) -> bool:
        i = 0
        while i < len(bits) - 1:
            if bits[i] == 1:
                i += 2
            else:
                i += 1
        return i == len(bits) - 1
```

```python
class Solution:
    def isOneBitCharacter(self, bits: List[int]) -> bool:
        i = 0
        while i < len(bits):
            if bits[i] == 1:
                i += 2
            else:
                i += 1
        return i == len(bits) - 1
```

(b) Credit assignment Qwen2.5-Coder-3B-Instruct w/ StructureCoder.

Figure 6: Credit assignment with different methods on Qwen2.5-Coder-3B-Instruct. The left is the correct response and the right is the incorrect response (error is introduced from the last token of Line 4). Each token is colored corresponding to the DPO implicit reward (darker is higher).



```python
def generate_diagonal(n, l):
    if l == 0:
        return []
    diagonal = [1]
    for i in range(1, l):
        diagonal.append(diagonal[-1] * (n + i) // i)
    return diagonal
```

```python
def generate_diagonal(n, l):
    if l == 0:
        return []
    diagonal = [1]
    for i in range(1, l):
        diagonal.append(diagonal[-1] * (n - i) // i)
    return diagonal
```

(a) Credit assignment Qwen2.5-Coder-7B-Instruct w/ DPO.

```python
def generate_diagonal(n, l):
    if l == 0:
        return []
    diagonal = [1]
    for i in range(1, l):
        diagonal.append(diagonal[-1] * (n + i) // i)
    return diagonal
```

```python
def generate_diagonal(n, l):
    if l == 0:
        return []
    diagonal = [1]
    for i in range(1, l):
        diagonal.append(diagonal[-1] * (n - i) // i)
    return diagonal
```

(b) Credit assignment Qwen2.5-Coder-7B-Instruct w/ StructureCoder.

Figure 7: Credit assignment with different methods on Qwen2.5-Coder-7B-Instruct. The left is the correct response and the right is the incorrect response (error is introduced in Line 6). Each token is colored corresponding to the DPO implicit reward (darker is higher).

Algorithm 1: Training Data Construction Pipeline

```
Input: $D = \{(q, c, t)\}$, where $q$ is a question, $c$ is the corresponding code, and $t$ are test cases
Output: Final training set ready for fine-tuning
foreach $(q, c, t) \in D$ do
    Parse $c$ into an Abstract Syntax Tree (AST);
    Extract target code blocks $M = \{m_1, m_2, \ldots, m_n\}$;
    foreach $m_i \in M$ do
        Extract prefix $p_i$ and suffix $s_i$;
        Construct FIM prompt: <PRE> Convert($q$) $p_i$ <SUF> $s_i$ <MID>;
        Generate $m$ completions $G_i = \{g_i^{(1)}, \ldots, g_i^{(m)}\}$;
        foreach $g_i^{(j)} \in G_i$ do
            $full\_code \leftarrow p_i + g_i^{(j)} + s_i$;
            Execute $full\_code$ on test cases $t$;
            if passes then
                Add $g_i^{(j)}$ to correct candidates;
            else
                Add $g_i^{(j)}$ to incorrect candidates;
            end
        end
    end
    foreach incorrect completion $g^-$ do
        Find closest correct $g^+$ by edit distance;
        Form preference pair $(g^+, g^-)$;
    end
end
foreach preference pair $(g^+, g^-)$ do
    Sample prompt format using Bernoulli distribution ($\alpha$);
    if format = FIM then
        Positive $\leftarrow$ <PRE> Convert($q$) $p_i$ <SUF> $s_i$ <MID> $g^+$;
        Negative $\leftarrow$ <PRE> Convert($q$) $p_i$ <SUF> $s_i$ <MID> $g^-$;
    else
        Positive $\leftarrow$ User: $q$; Assistant: $p_i + g^+ + s_i$;
        Negative $\leftarrow$ User: $q$; Assistant: $p_i + g^- + s_i$;
    end
    Add sample to training set;
end
Sort training samples by length of $g^+$ (short $\rightarrow$ long);
return Final training set;
```