Teaching Your Models to Understand Code via Focal Preference Alignment

Jie Wu[⋆ϕ], **Haoling Li**[⋆ϕ], **Xin Zhang**[◇⋆π], **Xiao Liu**[π], **Yangyu Huang**[π], **Jianwen Luo**[σ], **Yizhen Zhang**[ϕ], **Zuchao Li**[◇], **Ruihang Chu**[†ϕ], **Yujiu Yang**[ϕ], **Scarlett Li**[π]
[ϕ]Tsinghua University [π]Microsoft Research [σ]CASIA [◇]Wuhan University

Abstract

Preference learning extends the performance of Code LLMs beyond traditional supervised fine-tuning by leveraging relative quality comparisons. In existing approaches, a set of n candidate solutions is evaluated based on test case success rates, with the candidate demonstrating a higher pass rate being labeled as positive and its counterpart with a lower pass rate as negative. However, because this approach aligns entire failing code blocks rather than pinpointing specific errors, it lacks the granularity necessary to capture meaningful error-correction relationships. As a result, the model is unable to learn more informative error-correction patterns. To address these issues, we propose Target-DPO, a new preference alignment framework that mimics human iterative debugging to refine Code LLMs. Target-DPO explicitly locates error regions and aligns the corresponding tokens via a tailored DPO algorithm. To facilitate it, we introduce the CodeFlow dataset, where samples are iteratively refined until passing tests, with modifications capturing error corrections. Extensive experiments show that a diverse suite of Code LLMs equipped with Target-DPO achieves significant performance gains in code generation and improves on challenging tasks like BigCodeBench. In-depth analysis reveals that Target-DPO yields fewer errors. Code, model and datasets are in: https://github.com/JieWu02/Target-DPO.

1 Introduction

Preference learning offers a promising complement to supervised fine-tuning (SFT) (Zhang et al., 2023) for improving code generation accuracy in coding large language models (Code LLMs). Existing methods (Zhang et al., 2024, 2025; Liu et al., 2024c) mainly rely on unit test feedback to construct preference pairs. In these approaches, a Code LLM generates multiple code snippets as candidates and evaluates each against a suite of test cases. The snippet with the higher pass rate is considered preferred, while the one with the lower pass rate is marked as dispreferred, which forms the pair for preference learning such as Direct Preference Optimization (DPO) (Rafailov et al., 2023). However, this paradigm suffers from two critical drawbacks. First, constructing pairs purely on pass rate cannot guarantee high-quality labels. A high-pass-rate snippet may still carry subtle but crucial bugs, while a low-pass-rate snippet might need only a few modifications to become correct (as shown in Figure 2), resulting in noisy preference data. Second, as errors may be isolated to specific code parts, aligning entire snippets can dilute the correct sig-

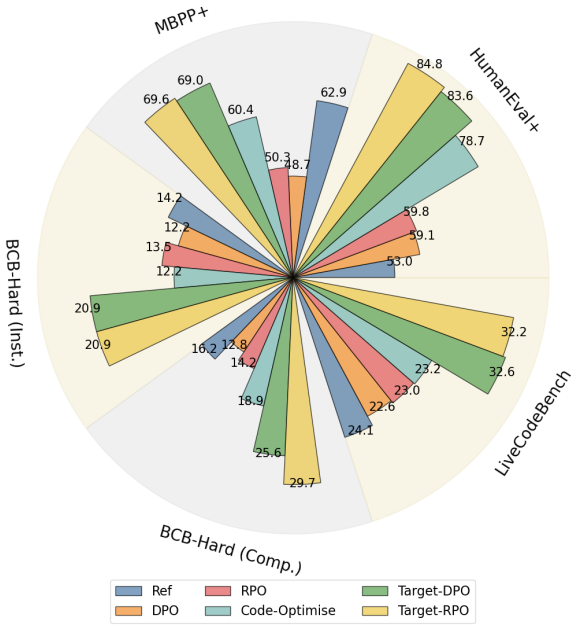

Figure 1: Target-DPO achieves significant performance gains over DPO variants on challenging coding tasks, *i.e.*, BigCodeBench-Hard, with Qwen2.5-Coder-7B.

[⋆]Equal contribution. Work done during the internships of Jie Wu, Haoling Li, and Jianwen Luo at Microsoft Research. Email: {wujie24,li-hl23}@mails.tsinghua.edu.cn.
[◇]Project leader.
[†]Corresponding author.

nal. It forces the model to adjust irrelevant tokens and hinders its ability to learn more specific error patterns (Pal et al., 2024; Chen et al., 2024; Wu et al., 2024), which would increase the overfitting risk. The limitations call for a better framework that can pinpoint error regions and apply targeted learning to correct those precise areas.

To tackle these challenges, we draw inspiration from how developers debug code. Typically, a programmer first locates the module that generates errors based on execution feedback and then focuses on fixing that specific portion until all tests pass. Following this human approach, we introduce Target-DPO, a novel framework for preference learning in Code LLMs that leverages iterative debugging insights. Rather than only using pass rate to measure the degree of preference, Target-DPO derives preference pairs from debugging process itself, where the refine steps yield a preference pair, labeling the corrected snippet as preferred and its previous version as dispreferred. By explicitly contrasting the tokens that resolve the error, Target-DPO trains Code LLMs to learn fine-grained alignment for precise error correction, enabling models to truly understand the code.

To support this framework, we efficiently synthesize high-quality preference pairs to create CodeFlow, a novel dataset systematically recording code iterations and corresponding error corrections. Compared to sampling-based methods, CodeFlow enables the efficient creation of preference pairs by (1) generating code snippets and test cases, (2) iteratively refining code until all tests pass, and (3) annotating key token changes between failed and corrected versions. This process ensures that preference learning focuses on the actual error-resolution steps taken by developers.

Building on CodeFlow, we propose an improved DPO algorithm that rewards correct code tokens while penalizing only error-specific tokens in dispreferred samples, minimizing irrelevant noise during preference learning and thus improving efficiency. Comprehensive ablation studies show how to best select dispreferred samples and how much context to include during alignment, verifying our optimal design for code correction.

We conduct extensive experiments on five public datasets to validate the effectiveness of Target-DPO. With only 59k preference pairs, Target-DPO achieves significant performance gains across various base and instruct-tuned Code LLMs. Notably, as shown in Figure 1, Target-DPO attains superior results on complex coding tasks like BigCodeBench. Through detailed ablation studies, we also demonstrate that Target-DPO outperforms alternative strategies by a clear margin. Our contributions are as follows:

1. We propose Target-DPO, a novel framework that leverages the idea of iterative debugging to address challenges in preference learning, enabling more precise alignment on critical error tokens.

2. We construct a new function-level dataset CodeFlow that iteratively tracks token differences across preference pairs, and propose a tailored adaptation of the DPO algorithm that avoids unnecessary optimization noise.

3. Target-DPO consistently improves performance across diverse benchmarks and various base and instruct-tuned Code LLMs.

2 Target-DPO Framework

The Target-DPO framework mimics human iterative debugging to refine Code LLMs. It explicitly identifies error regions and focuses on aligning the corresponding tokens through a tailored DPO algorithm. To achieve this, Target-DPO follows two steps: (1) synthesizing preference code pairs through an iterative debugging process and locate error regions within code, resulting in the creation of CodeFlow (Section 2.1), and (2) performing fine-grained and focal alignment by contrasting critical tokens via the designed DPO algorithm (Section 2.2). The overall workflow is illustrated in Figure 3. In the following sections, we will provide a detailed description of each step.

2.1 Synthesize Preference Code Snippets

As part of our method, we synthesize 59k preference pairs to enable targeted alignment. In contrast to previous methods that synthesize preference pairs based on pass rate, Target-DPO synthesizes preference code snippets from an iterative debugging process. In this process, an initial code snippet is refined until it passes the test cases, and a preference pair is constructed between the final correct version and the previous iteration.

Generate Raw Code Snippets and Tests. To obtain diverse and complex code data, we adopt the practice of EpiCoder (Wang et al., 2025a), utilizing its feature tree-based synthesis framework with

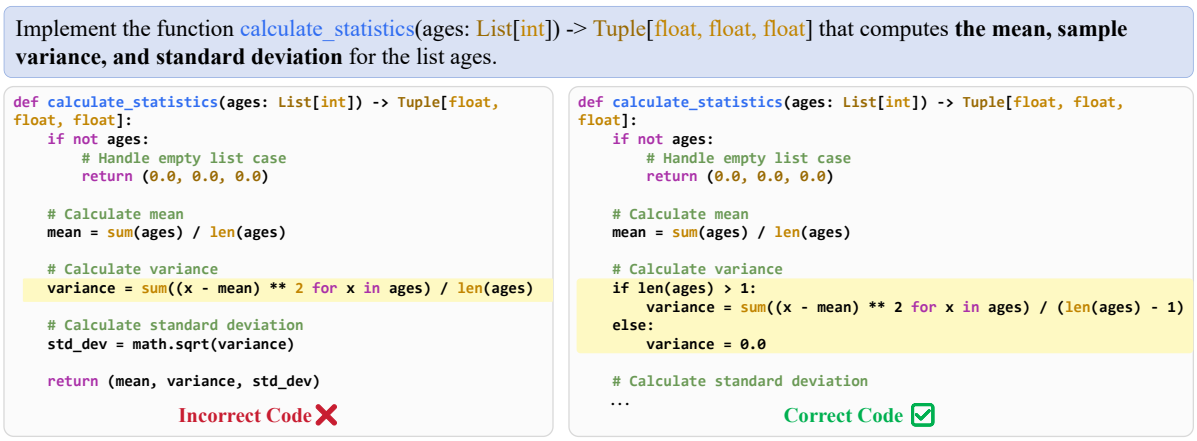

Figure 2: In LLM-generated code, errors are usually confined to critical parts. Minor adjustments to the corresponding erroneous tokens can correct the code while leaving the majority unchanged. Therefore, an effective error correction requires first identifying the key error lines and then performing focal alignment.

GPT-4o (OpenAI, 2024) to generate high-quality code and test cases. This approach directs the LLM to produce a coding task instruction, the corresponding code snippet, and multiple test cases.

To ensure the quality of the generated test cases, we applied several validation measures, including coverage analysis, LLM-based evaluation, and human verification. Detailed discussions and prompting examples are provided in Appendix A.2.

Iterative Refinement via Verification. LLMs cannot guarantee the correctness of generated code (Ma et al., 2025). Therefore, we verify each code sample and refine it through iterative debugging based on execution feedback from verified test cases. As shown in Figure 3, when the initial code fails the unit tests, we collect the error information and refine the code iteratively until it passes the test. The pass rate at the T-th iteration is reported in Figure 6 of Appendix A.1.

Our goal is to collect program changes made during the iterative debugging process. To reduce costs, we discard code that fails to be corrected within five iterations. Although this filter removes some extremely challenging cases, it does not make the generated dataset predominantly easier, as clarified in Appendix A.1. Samples requiring more than five iterations for a solution generally fail to pass all test cases anyway, regardless of additional sampling efforts.

After iterative debugging, we treat the final correct code as the preferred sample and randomly select an earlier version as the dispreferred sample, forming the pair (y^+, y^-) for preference learning.

Critical Difference Extraction. To identify the targeted regions for alignment, we extract the critical differences responsible for the functional divergence between each (y^+, y^-) pair. Specifically, we pinpoint the sets of differing lines, D^+ and D^-, by computing the Longest Common Subsequence (LCS). Lines not part of the LCS are considered difference lines, as detailed in Algorithm 1. Consequently, the key modifications distinguishing the preferred from the dispreferred samples are localized within D^+ and D^-. These segments encapsulate the changes driving the functional distinctions and represent the critical regions that Target-DPO aims to contrast and align.

Algorithm 1 Extracting Code Difference

Require: Code pair y^+ and y^-
Ensure: Difference lines $\mathcal{D}^+$ and $\mathcal{D}^-$ for y^-
1: Split y^+ and y^- into lines: y^+_{lines} and y^-_{lines}
2: Find the LCS of lines between y^+_{lines} and y^-_{lines}
3: Initialize $\mathcal{D}^+ = \emptyset$ and $\mathcal{D}^- = \emptyset$
4: $\mathcal{D}^+ = \{l^+ \in y^+_{\text{lines}} \mid l^+ \notin \text{LCS}\}$
5: $\mathcal{D}^- = \{l^- \in y^-_{\text{lines}} \mid l^- \notin \text{LCS}\}$
6: **return** $\mathcal{D}^+$ and $\mathcal{D}^-$

Quality Control for Preference Pairs. We initially synthesize 104k instruction data points through iterative refinement. To further ensure data quality, we implement several filtering measures. Specifically, rule-based filtering is applied to remove trivial or uninformative samples, such as those where: (i) D^- consists only of comments; (ii) D^- exceeds 20 lines; (iii) y^+ or y^- exceeds 2048 tokens; and (iv) Samples where the abstract syntax tree (AST) of y^+ and y^- are identical.

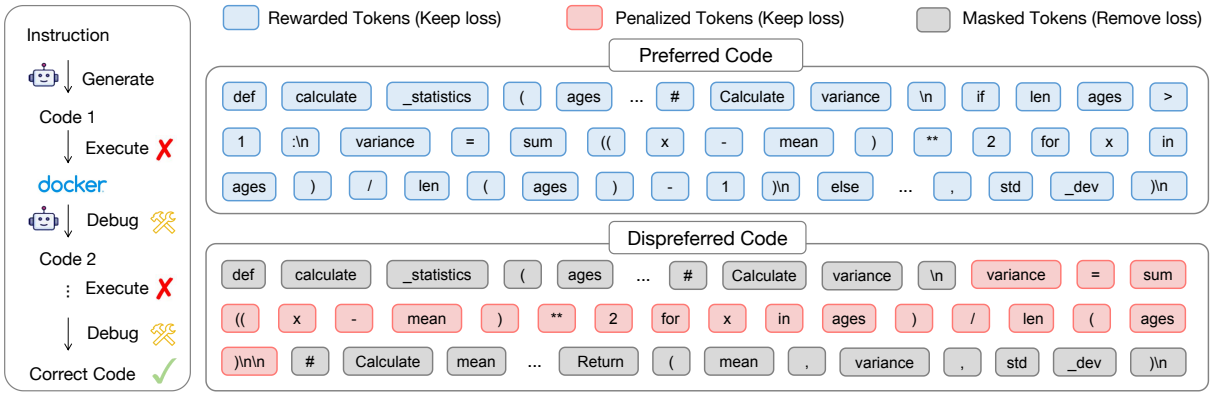

Figure 3: **Method Overview.** Target-DPO constructs preference pairs via iterative debugging, treating the correct version as preferred and the previous as dispreferred. DPO adaptations enable code LLMs to learn the correct pattern from the preferred code while highlighting critical tokens with a masking strategy in the dispreferred sample.

The application of these filters reduced the dataset to 84k samples. In the next stage, we utilized GPT-4o as an LLM-judge to assess whether a significant logical distinction existed between y^+ and y^-. We further filter out pairs where the differences are limited to code formatting, comments, variable names, whitespace, or blank lines. These efforts ensure that the selected and rejected samples reflect key functional differences. The final dataset, consisting of 59k samples, is thus prepared for preference-based alignment training.

2.2 Targeted Preference Alignment

Direct Preference Optimization (DPO) directly optimizes the policy model using relative quality comparisons. Given a prompt x, a preference pair (y^+, y^-), where y^+ is of higher quality than y^-, DPO aims to maximize the probability of the preferred response y^+ while minimizing that of the less desirable response y^-. The KL divergences for y^+ and y^- are defined as:

$$\mathcal{K}^+ = \log \frac{\pi_\theta(y^+|x)}{\pi_{\text{ref}}(y^+|x)}, \quad \mathcal{K}^- = \log \frac{\pi_\theta(y^-|x)}{\pi_{\text{ref}}(y^-|x)}, \tag{1}$$

and the optimization objective $\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}})$ is:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}} \left[\log \sigma \left(\beta \left(\mathcal{K}^+ - \mathcal{K}^-\right)\right)\right] \tag{2}$$

DPO optimizes the expectation over the pairwise preference dataset $\mathcal{D}$, and σ is the sigmoid function.

While Direct Preference Optimization (DPO) has demonstrated effectiveness in domains such as mathematics (Lai et al., 2024), its standard objective function, as shown in Equation (2), may be suboptimal for preference-based alignment in code generation, as a large portion of the tokens in y^+ and y^- are identical, with only minor differences. This can confuse the policy model in identifying the critical differences necessary for functional correctness, and diminish alignment gains (Pal et al., 2024; Chen et al., 2024; Wu et al., 2024).

To help code LLMs better grasp the critical tokens driving functional differences between preference pairs, we modify the DPO algorithm to highlight key tokens in the dispreferred code snippet using a masking strategy. Specifically, given $y^- = [y_1^-, y_2^-, .., y_L^-]$ containing L tokens, vanilla DPO computes $\mathcal{K}^-$ as:

$$\begin{aligned} K^- = \log \frac{\pi_\theta(y^-|x)}{\pi_{\text{ref}}(y^-|x)} &= \log \frac{\prod_{i=1}^{L} \pi_\theta(y_i^-|x)}{\prod_{i=1}^{L} \pi_{\text{ref}}(y_i^-|x)} \\ &= \sum_{i=1}^{L} \log \frac{\pi_\theta(y_i^-|x)}{\pi_{\text{ref}}(y_i^-|x)} \end{aligned} \tag{3}$$

We make the following adaptations to $\mathcal{K}^-$ while keeping $\mathcal{K}^+$ unchanged:

$$K^{+\prime} = K^+ \tag{4}$$

$$K^{-\prime} = \textstyle\sum_{i=1}^{L} \mathbb{I}(y_i^- \in \mathrm{D}^-) \log \frac{\pi_\theta(y_i^-|x)}{\pi_{\text{ref}}(y_i^-|x)} \tag{5}$$

Equation (4) guides the code LLM to learn correct code generation patterns from y^+. In contrast, Equation (5) explicitly focuses on contrasting critical tokens within y^-. It achieves this by masking tokens in the dispreferred code that do not appear in D^-, thereby excluding correct tokens in y^- from the loss computation, as illustrated on the right side of Figure 3. By penalizing critical tokens that cause functional errors and preventing over-optimization on tokens common to both y^+ and y^-, Target-DPO achieves a more fine-grained alignment tailored for code, improving upon previous sample-level optimization approaches. This

refined strategy enables code LLMs to better internalize correct coding patterns and more effectively identify crucial token-level errors.

Our loss also targets pairwise optimization:

$$\mathcal{L}'_{\text{DPO}} = -\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}} \log \sigma \left(\beta \left(\mathcal{K}^{+\prime} - \mathcal{K}^{-\prime}\right)\right) \quad (6)$$

Correspondingly, the RPO loss (Liu et al., 2024a; Pang et al., 2024), a variant of DPO, consists of a weighted SFT loss on y^+, scaled by α. Our modified DPO loss also complements RPO, and the RPO-format $\mathcal{L}'_{\text{RPO}}$ loss is:

$$\mathcal{L}_{\text{SFT}} = -\mathbb{E}_{(x,y^+)\sim\mathcal{D}} \left[\log p_\theta(y^+|x)\right] \quad (7)$$

$$\mathcal{L}_{\text{RPO}'} = \mathcal{L}'_{\text{DPO}} + \alpha\mathcal{L}_{\text{SFT}} \quad (8)$$

3 Experiments

Experiment Setup. For our Target-DPO, the learning rate is set to 1e-5 for the 7B code LLMs and 5e-6 for the 15B models, using a global batch size of 128, with a cosine scheduler and warm-up. The maximum sequence length is set to 2048 tokens. Detailed training settings are presented in Appendix A.3 For the DPO algorithm, β is set to 0.1, and for RPO, α is set to 1.0. The rationale behind the choice of α and β is supported by ablation studies presented in Appendix C.2. π_θ and π_{ref} are both initialized with the weights of the evaluated model, while π_{ref} keeps frozen during training.

Benchmarks. We evaluate the Code LLMs using multiple benchmarks: HUMANEVAL Base (Chen et al., 2021), HUMANEVAL Plus (Liu et al., 2023), Mostly Basic Python Problems (MBPP Base (Austin et al., 2021), MBPP Plus), LiveCodeBench (LCB) (Jain et al., 2024) (v5 with problems released between May 2023 and Jan 2025), and BIG-CODEBENCH (BCB) (Zhuo et al., 2025) with instruct and completion splits. We report the pass@1 score under greedy decoding.

Evaluated Models and Baselines. We evaluate models including DeepSeek-Coder-7B-Instruct-v1.5 (Guo et al., 2024), CodeQwen1.5-7B-Chat (Bai et al., 2023), as well as base models such as Qwen2.5-Coder-7B (Hui et al., 2024b) and StarCoder2-15B (Lozhkov et al., 2024). Results for the 32B model are in Appendix C.1. CodeDPO and PLUM are compared using their reported results, as their data and code are currently unavailable. Code-Optimise (Gee et al., 2025) is reproduced using GPT-4o, with 100 solutions sampled at a temperature of 0.6 for each problem. The DPO-PvF setting results for Code-Optimise are reported.

4 Main Results

Table 1 presents a comparison between baseline models, DPO variants, and Target-DPO. We discuss the findings from the following perspectives.

Focal Alignment Outperforms Global Alignment. Preference pairs from iterative debugging differ significantly from those in datasets like Code-Optimise, causing a performance drop with vanilla DPO or RPO. While some settings, like DS-Coder-7B-Instruct-DPO, show gains on BIG-CODEBENCH, DPO and RPO generally underperform compared to baselines. In typical correction scenarios, an LLM modifies only a small portion of the code to fix errors, creating highly similar preference pairs. This overlap introduces ambiguity, as identical tokens in both positive and negative examples weaken the model's ability to distinguish meaningful differences.

This degradation underscores the need for explicit mechanisms to focus the policy model on tokens responsible for functional faults. Target-DPO addresses this by emphasizing error tokens in the dispreferred code and explicitly contrasting the critical edits. As shown in Table1, Target-DPO outperforms DPO by 3.3% and 5.9% on average across benchmarks, while Target-RPO yields improvements ranging from 6.3% to 12.4%.

Target-DPO Achieves Significant Improvements over Methods that Rely on Coarse-grained Pass Rate Signals. While methods like PLUM and CodeDPO, which construct preference pairs by testing multiple sampled solutions, offer a straightforward and effective approach, their reliance on coarse-grained pass/fail signals inherently limits the model's ability to learn nuanced error correction and generalize improvements. As shown in

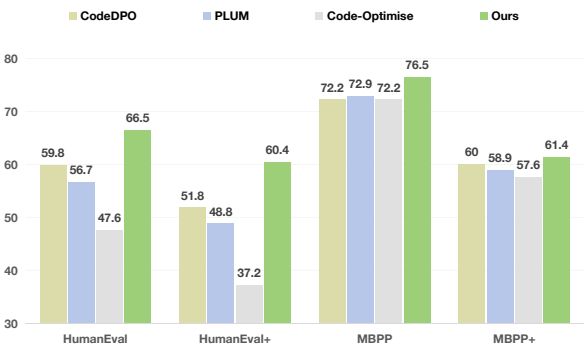

Figure 4: Comparison with CodeDPO, PLUM, and Code-Optimise using DeepSeekCoder-6.7B. Additional results are provided in Appendix B.3.

Model	Variant	HumanEval		MBPP		BCB-Full		BCB-Hard		LCB	Avg.
		Base	*Plus*	*Base*	*Plus*	*Comp.*	*Inst.*	*Comp.*	*Inst.*	*Inst.*	
DS-Coder-7B-Ins-v1.5	Ref.	75.6	71.3	75.2	62.3	43.8	35.5	15.5	10.1	20.6	45.5
	DPO	69.5	65.2	77.2	67.2	46.1	37.9	12.2	14.2	20.4	45.5
	RPO	65.2	59.8	75.7	66.1	43.2	37.5	10.8	13.8	20.2	43.6
	Code-Optimise	64.6	60.4	78.8	**69.3**	45.2	36.5	13.5	13.5	21.3	44.8
	Target-DPO	76.2	72.0	**79.1**	65.3	47.5	37.8	**22.3**	17.6	21.8	48.8
	Target-RPO	**78.0**	**73.2**	78.8	67.2	**49.3**	**39.0**	20.9	**20.9**	**22.0**	**49.9**
CodeQwen1.5-7B-Chat	Ref.	83.5	78.7	79.4	69.0	43.6	39.6	15.5	**18.9**	15.3	49.3
	DPO	79.3	73.8	79.9	69.0	43.3	36.1	14.9	10.8	15.5	47.0
	RPO	79.3	73.2	80.2	68.8	41.6	32.5	14.8	10.6	12.9	46.0
	Code-Optimise	78.5	75.0	80.7	69.6	43.3	36.1	17.6	11.5	16.2	47.6
	Target-DPO	89.6	85.4	**83.9**	69.8	**48.7**	**39.9**	20.3	16.9	**18.1**	**52.5**
	Target-RPO	**89.6**	**86.0**	82.5	**70.4**	48.4	38.3	**20.3**	18.2	17.2	52.3
StarCoder2-15B	Ref.	46.3	37.8	66.2	53.1	38.4	-	12.2	-	-	-
	DPO	51.8	45.1	63.8	42.9	27.3	16.2	8.1	5.4	12.7	30.4
	RPO	53.0	45.7	63.0	42.6	28.7	17.2	9.1	6.0	13.1	30.9
	Code-Optimise	61.0	54.9	66.5	53.4	31.8	18.8	6.8	6.1	14.9	34.9
	Target-DPO	70.7	64.6	**67.2**	**54.5**	39.7	37.7	17.6	16.9	18.7	43.1
	Target-RPO	**73.2**	**65.2**	65.9	53.4	**40.3**	**38.8**	**18.9**	**18.2**	**19.4**	**43.7**
Qwen2.5-Coder-7B	Ref.	61.6	53.0	76.9	62.9	45.8	40.2	16.2	14.2	24.1	43.9
	DPO	71.3	59.1	76.2	48.7	38.8	28.5	12.8	12.2	22.6	41.1
	RPO	71.3	59.8	70.9	50.3	39.8	29.7	14.2	13.5	23.0	41.4
	Code-Optimise	82.3	78.7	76.2	60.4	48.5	39.6	18.9	12.2	23.2	48.9
	Target-DPO	89.0	83.6	83.1	69.0	52.7	41.0	25.6	20.9	32.6	55.3
	Target-RPO	**89.6**	**84.8**	**83.3**	**69.6**	**53.3**	**43.1**	**29.7**	**20.9**	**32.2**	**56.3**

Table 1: Pass@1 (%) results of different LLMs on HumanEval, MBPP, BigCodeBench, and LiveCodeBench-v5 (LCB) under greedy decoding setting. We conducted the evaluation on the Full and Hard subsets of BigCodeBench (BCB), including the Complete (Comp.) and Instruct (Inst.) tasks. The best results are highlighted in Bold.

Figure 4, this limitation becomes apparent when compared to our Target-DPO.

Target-DPO Improves Challenging Coding Task. We highlight that the Target-DPO framework has the potential to boost Code LLMs to solve complex coding tasks. Notably, Qwen2.5-Coder-7B equipped with Target-DPO achieves a 29.7% pass@1 score on BigCodeBench Complete Hard, matching the performance of larger Code LLMs DeepSeek-Coder-V2-Instruct (29.7%) and Claude-3-Opus (29.7%) (Anthropic, 2024), and approaching Llama-3.1-405B-Instruct (30.4%) (Grattafiori et al., 2024). When given more attempts, Target-DPO achieves pass@5 of 45.7%, outperforming DeepSeek-R1 (40.5%) (DeepSeek-AI et al., 2025) and GPT-o1 (40.2%). On the Instruct Hard split, pass@5 of the Target-DPO-Qwen is 34.7%, comparable to the performance of GPT-o3-mini (33.1%).

5 Ablation Study

Despite the effectiveness of Target-DPO in pinpointing critical error regions, there remain open questions about how best to incorporate negative examples and how much context is truly beneficial for code correction. We therefore explore several settings: (i) **SFT**: Supervised fine-tuning using the positive sample from the preference pair; (ii) **Hybrid Training**: Half of the samples in a batch are trained using vanilla DPO, while the other half follows the Target-DPO approach; (iii) **Diff-Augmentation**: provide more context for the dispreferred sample by including 1 or 2 lines of tokens before and after D^-; and (iv) **Symmetric Masking Strategy**: The Code LLMs learn from the tokens in D^+ rather than the full sequence of positive sample. In Figure 5, we illustrate these settings.

Supervised Fine-Tuning. A comparison with EpiCoder-SFT, considering varying amounts of training data, is shown in Table 2. Our Target-DPO achieves performance comparable to the strong SFT baseline EpiCoder-380k using only 59k training samples (about one-sixth), unveiling the power of targeted alignment.

The positive samples undergo iterative debugging and are validated by test cases, they maintain high quality, allowing SFT to achieve reasonably strong performance. However, SFT overlooks dispreferred samples, missing the opportunity to contrast and precisely align positive and negative examples. In contrast, Target-DPO not only lever-

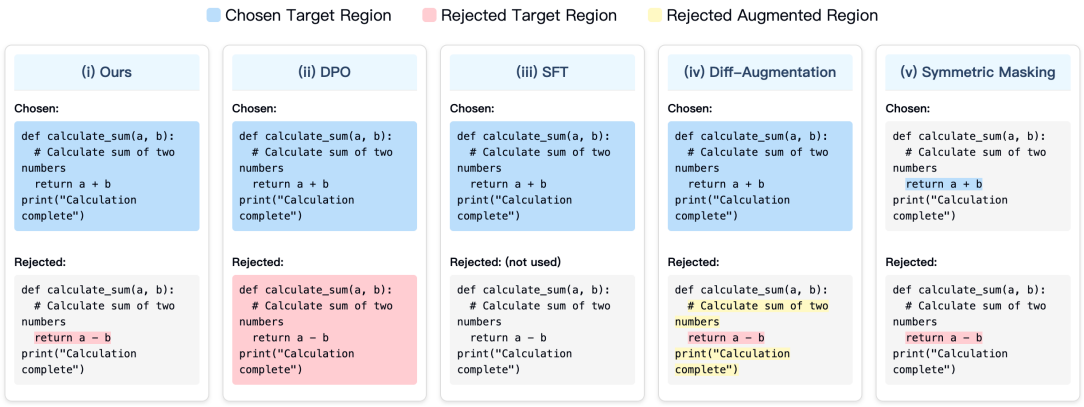

Figure 5: Illustration for Target-DPO and its ablations. Target-DPO rewards correct code tokens while penalizing only error-specific tokens in rejected code, teaching models to truly understand code through targeted alignment.

	HumanEval(Avg)	MBPP(Avg)	BCB(Complete)	BCB(Instruct)	Average
SFT (EpiCoder 40k)	83.9	75.5	50.9	39.1	62.4
SFT (EpiCoder 80k)	85.1	**78.9**	52.3	39.4	63.9
SFT (EpiCoder 380k)	85.7	77.8	**53.4**	**43.8**	**65.2**
SFT (CodeFlow 59k)	85.5	75.7	51.6	39.1	62.9
Our Target-DPO (59k)	**87.2**	76.5	53.3	43.1	65.0

Table 2: Results of EpiCoder-SFT with varying amounts of training data and our method on Qwen2.5-Coder-7B.

	Aug	Hybrid	HumanEval		MBPP		BCB-Inst	BCB-Comp	Average
			Base	*Plus*	*Base*	*Plus*			
CodeQwen1.5-7B-Chat	-	-	83.5	78.7	79.4	69.0	39.6	43.6	65.6
Target-DPO	✗	✔	83.5	79.3	81.5	66.1	38.2	44.5	65.5
	✔	✗	83.5	76.8	80.2	65.1	34.7	44.5	64.1
	✗	✗	**89.6**	85.4	**83.9**	69.8	**39.9**	**48.7**	**69.6**
Target-RPO	✗	✔	84.8	79.3	81.5	67.7	34.6	44.0	65.3
	✔	✗	86.0	79.9	81.5	65.9	36.0	45.0	65.7
	✗	✗	**89.6**	**86.0**	82.5	**70.4**	38.3	48.4	69.2

Table 3: Ablation study on how much contextual information from negative examples is beneficial for Target-DPO, evaluated using CodeQwen1.5-7B-Chat. Additional results with Qwen2.5-Coder-7B are provided in Appendix C.

	HumanEval(Avg)	MBPP(Avg)	BCB(Complete)	BCB(Instruct)	Avg.
SFT (Correct)	**85.5**	**75.7**	**51.6**	**39.1**	**63.0**
SFT (Incorrect)	82.7	73.5	49.2	38.6	61.0

Table 4: Performance comparison of SFT on correct and incorrect code from CodeFlow uisng Qwen2.5-Coder-7B.

ages error-free code to increase the likelihood of correct code but also precisely penalizes tokens responsible for critical errors, achieving finer-grained alignment and better performance consequently.

Hybrid Training & Diff-Augmentation. Both settings expose Code LLMs to more tokens from the dispreferred samples but differ in scope: in Hybrid Training, 50% of the training samples use the entire dispreferred sequence, while Diff-Augmentation provides a small token window around the D^-. Table 3 shows that while adding extra context around D^- may appear beneficial, it often introduces noise that confuses the model, making it unclear which parts need local alignment, ultimately leading to degraded performance.

We find that concentrating solely on the most critical tokens yields better results, highlighting the importance of accurately grounding these tokens for more effective targeted alignment. The iterative debugging process naturally supports this precise localization, as typically only a small portion of the code changes between iterations, while the majority

remains unchanged. These targeted regions can be easily identified using the Longest Common Subsequence (LCS), allowing meaningful differences to be isolated with high precision.

Symmetric Masking Strategy. When training with the symmetric masking, where Code LLMs learn from both D^+ and D^- without access to the full positive sample, the model struggles to retain its core code generation capabilities and fails to benchmark effectively. The primary goal of Code LLMs is to generate complete and correct code. Although learning symmetrically from both D^+ and D^- may seem appealing, the focus should be on ensuring Code LLMs learn from fully correct code rather than fragmented pieces. Without complete code contexts, the positive sample cannot properly align with the instruction, leading to incomplete and misleading signals in the learning process.

Generated Test Cases can Distinguish Good and Error Code. Table 4 compares supervised fine-tuning using either preferred or dispreferred samples. The results show that SFT on preferred samples outperforms that on dispreferred ones by an average of 2.0%. This quality gap between pairs, introduced through debugging iterations, suggests that test cases effectively differentiate high- and low-quality code snippets, providing training pairs with clear quality contrast. In our debugging pattern, the quality differences between code snippets are primarily influenced by the feedback from test cases, indicating that test cases can reliably distinguish between good and bad code when verified through dedicated efforts.

6 Related Work

Code Language Models. Powerful Code LLMs like Qwen2.5-Coder (Hui et al., 2024a), DeepSeek-Coder (Guo et al., 2024), StarCoder (Li et al., 2023; Lozhkov et al., 2024), Magicoder (Wei et al., 2024) and EpiCoder (Wang et al., 2025b) demonstrate their capabilities in various code generation tasks. Current Code LLMs primarily focus on supervised fine-tuning during the post-training stage. While SFT enables Code LLMs to learn the correct patterns, it fails to effectively make them aware of incorrect patterns or how to rectify errors in code. In this work, Target-DPO framework aims to enable Code LLMs to further learn through pairwise contrasting of critical tokens (Lin et al., 2024), allowing Code LLMs to continually improve.

Reinforcement Learning (RL) (Hu et al., 2025; Kaufmann et al., 2024) maximizes the following objective for a prompt x and response y:

$$\max_{\pi_\theta} \mathbb{E}_{x\sim D_p, y\sim\pi_\theta(\cdot|x)} \left[r(x,y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} \right]$$

where D_p is the dataset, π_θ is the policy model to be optimized, π_{ref} is the reference, and β controls the degree of regularization. RL for code generation attracts attention recently (Dou et al., 2024; Li et al., 2024; Sun et al., 2024; Miao et al., 2024; Dai et al., 2025). A commonly used approach is DPO (Rafailov et al., 2023), which eliminates the need for an explicit reward model r. Variants like RPO (Liu et al., 2024a; Pang et al., 2024) and KTO (Ethayarajh et al., 2024) are also frequently used in optimizing code generation.

Preference Pair Construction. Existing methods construct preference pairs by ranking candidate solutions based on pass rates. PLUM (Zhang et al., 2024) constructs preference pairs by ranking candidate code solutions based on passed test cases. Code-Optimise (Gee et al., 2025) incorporates efficiency as an additional learning signal, augmented with annotations from unit test feedback and execution time. AceCoder (Li et al., 2024) selects pairs with distinct pass rate differences. DSTC (Liu et al., 2024b) constructs preference pairs using self-generated code and tests. A related concurrent work is CodeDPO, which formulates preference learning as a direct optimization problem using pass/fail signals and proposes a PageRank-inspired algorithm to select high-quality preference pairs. In contrast, our method aligns code LLMs through error-resolving edits rather than relying on coarse-grained execution outcomes. This fine-grained supervision provides richer training signals that better capture the semantics of code correction, resulting in improved performance, as verified in Figure 4.

7 Conclusion

We present Target-DPO, a novel preference alignment framework that emulates human iterative debugging to capture critical errors in incorrect code for precise optimization. Target-DPO first identifies error-prone regions and applies an improved DPO algorithm contrasting pivotal segments, teaching Code LLMs to understand and correct code through targeted preference alignment, achieving promising coding performance. To support this framework, we efficiently synthesize high-quality preference pairs to create CodeFlow, where

each sample undergoes iterative refinement until it passes unit tests, with the modification history providing a natural record of error corrections. Extensive experiments show that Target-DPO-equipped Code LLMs achieve significant performance improvements in code generation and excel in tackling basic and complex coding tasks.

Limitations

Target-DPO is inspired by the debugging pattern of developers, serving as a novel framework for fine-grained preference learning in Code LLMs. Instead of using pass rate alone, Target-DPO derives preference pairs from iterative debugging process. By contrasting critical tokens between a corrected version and its preceding iteration, Target-DPO helps the model to understand code through targeted alignment. However, this study focuses on a dataset of 59k samples without further expansion, which may limit generalizability, but offers opportunities for future exploration with larger data.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China (Grant No. 62576191), the research grant No. CT20240905126002 of the Doubao Large Model Fund, and the National Natural Science Foundation of China (No. 62306216).

References

Anthropic. 2024. Claude 3.5: Advancing ai safety and performance. Technical report, Anthropic.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, and et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Huayu Chen, Guande He, Hang Su, and Jun Zhu. 2024. Noise contrastive alignment of language models with explicit rewards. *CoRR*, abs/2402.05369.

Mark Chen, Jerry Tworek, Heewoo Jun, and Qiming Yuan et al. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2025. Process supervision-guided policy optimization for code generation.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, and et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, and Limao et al. Xiong. 2024. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, Bangkok, Thailand. Association for Computational Linguistics.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. Model alignment as prospect theoretic optimization. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2025. Code-optimise: Self-generated preference data for correctness and efficiency. *Preprint*, arXiv:2406.12502.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

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

Yulan Hu, Ge Chen, Jinman Zhao, Sheng Ouyang, and Yong Liu. 2025. Coarse-to-fine process reward modeling for mathematical reasoning. *Preprint*, arXiv:2501.13622.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, and et al. 2024a. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024b. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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

Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. 2024. A survey of reinforcement learning from human feedback. *Preprint*, arXiv:2312.14925.

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. *Preprint*, arXiv:2406.18629.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. Acecoder: An effective prompting technique specialized in code generation. *ACM Trans. Softw. Eng. Methodol.*, 33(8).

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, and et al. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Zicheng Lin, Tian Liang, Jiahao Xu, Xing Wang, Ruilin Luo, Chufan Shi, Siheng Li, Yujiu Yang, and Zhaopeng Tu. 2024. Critical tokens matter: Token-level contrastive estimation enhence llm's reasoning capability. *arXiv preprint arXiv:2411.19943*.

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 *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Zhihan Liu, Miao Lu, Shenao Zhang, Boyi Liu, Hongyi Guo, Yingxiang Yang, Jose Blanchet, and Zhaoran Wang. 2024a. Provably mitigating overoptimization in RLHF: Your SFT loss is implicitly an adversarial regularizer. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Zhihan Liu, Shenao Zhang, Yongfei Liu, Boyi Liu, Yingxiang Yang, and Zhaoran Wang. 2024b. Dstc: Direct preference learning with only self-generated tests and code to improve code lms. *Preprint*, arXiv:2411.13611.

Zhihan Liu, Shenao Zhang, and Zhaoran Wang. 2024c. DSTC: direct preference learning with only self-generated tests and code to improve code lms. *CoRR*, abs/2411.13611.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, and et al. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. *Preprint*, arXiv:2501.01054.

Yibo Miao, Bofei Gao, Shanghaoran Quan, Junyang Lin, Daoguang Zan, Jiaheng Liu, Jian Yang, Tianyu Liu, and Zhijie Deng. 2024. Aligning codellms with direct preference optimization. *Preprint*, arXiv:2410.18585.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Arka Pal, Deep Karkhanis, Samuel Dooley, Manley Roberts, Siddartha Naidu, and Colin White. 2024. Smaug: Fixing failure modes of preference optimisation with dpo-positive. *Preprint*, arXiv:2402.13228.

Richard Yuanzhe Pang, Weizhe Yuan, He He, Kyunghyun Cho, Sainbayar Sukhbaatar, and Jason E Weston. 2024. Iterative reasoning preference optimization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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*.

Jiankai Sun, Chuanyang Zheng, Enze Xie, Zhengying Liu, Ruihang Chu, and et al. 2024. A survey of reasoning with foundation models. *Preprint*, arXiv:2312.11562.

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, Jinsong Su, Qi Chen, and Scarlett Li. 2025a. Epicoder: Encompassing diversity and complexity in code generation. In *Arxiv*.

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, et al. 2025b. Epicoder: Encompassing diversity and complexity in code generation. *arXiv preprint arXiv:2501.04694*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Junkang Wu, Yuexiang Xie, Zhengyi Yang, Jiancan Wu, Jinyang Gao, Bolin Ding, Xiang Wang, and Xiangnan He. 2024. β-dpo: Direct preference optimization with dynamic β. *CoRR*, abs/2407.08639.

Dylan Zhang, Shizhe Diao, Xueyan Zou, and Hao Peng. 2024. PLUM: preference learning plus test cases yields better code language models. *CoRR*, abs/2406.06887.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2025. CodeDPO: Aligning code models with self generated and verified source code.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. 2023. Instruction tuning for large language models: A survey. *CoRR*, abs/2308.10792.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, and et al. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

Appendix

In this appendix, we first provide more details of our core methodology, including preference pair construction and implementation specifics (Section A). Section B then introduces the evaluation benchmarks and presents comprehensive experimental results, showcasing performance on various benchmarks and detailed comparisons against Epi-Coder and other relevant methods. Subsequently, we provide in-depth analyses such as scaling laws, ablation studies on key parameters, data diversity assessments, error pattern examinations, and efficiency evaluations (Section C).

A Methodology and Data Construction

This section details the core methodology of our proposed approach, including the iterative refinement process for preference pair construction and the generation and quality assessment of synthetic test data. Implementation specifics relevant to these methodological aspects are also covered.

A.1 Iterative Refinement and Preference Pair Construction

The core of our data generation relies on an iterative refinement process. Figure 6 illustrates the progression of code sample pass rates through successive refinement iterations using execution verification feedback.

As depicted in Figure 6, at the first attempt (iter0), only 36.7% of the code samples pass their corresponding test file, indicating that debugging is necessary for the remaining code. Failed codes go through continual refinement, with the pass rate gradually approaching 67.5%. The pass rate rises sharply from iter 1 to iter 3 and then slows. Between iter 4 and iter 5, only 1.2% of cases improve, indicating incremental benefits from further iterations. Thus, additional iterations are not considered.

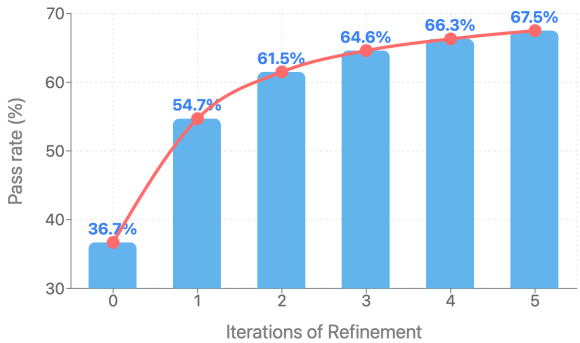

Figure 6: The pass rate progression across iterations of refinement with execution verification feedback.

Current methods construct preference pairs based on pass rate signals and conduct DPO to optimize Code LLMs. Two notable limitations arise: one from the data and one from the algorithm.

Regarding preference data, a snippet with a low pass rate may only require minor modifications to become correct, as errors tend to be isolated to specific parts of the code. We address this by iterative debugging with editing traces naturally annotated by the differences between iterations.

Regarding the algorithm, relying solely on full preference learning can introduce noise during optimization, as positive and negative pairs can be highly similar. This not only hinders the model from learning more effective error correction patterns but also increases the risk of overfitting. We solve this by explicitly identifying which parts of the code need to be aligned.

Iterative debugging can pass through 67.5% of tasks within a 5-time API call budget for each task. But given just 5 sampling attempts, the pass rate falls to 51.90% averaged across 5k samples, as shown in Table 5. This initial comparison highlights that iterative debugging can achieve a higher pass rate under similar API constraints.

	API Calls	**Pass rate** (%)
Debugging (Ours)	Up to 5 times	**67.5**
Sampling	5 times for each task	51.9

Table 5: Pass Rate of Debugging and Sampling for Preference Pair Construction.

To further investigate the limits of sampling for difficult cases, we collected 5k samples that didn't succeed within 5 debugging iterations. Table 6 details the pass rate when applying N sampling solutions to these difficult tasks.

The results in Table 6 indicate that for samples which could not succeed with 5 iterations using

Sampled Solutions N	**Passed**	**Failed**
5	2.42%	97.58%
10	3.26%	96.26%
15	3.96%	96.04%
30	4.20%	95.80%
50	4.36%	95.64%

Table 6: Pass Rate of Sampling for Difficult Cases (Failed within 5 Debugging Iterations).

interpreter feedback and runtime error information, additional sampling alone yields very low pass rates (e.g., only 3.96% pass with 15 sampling attempts). This suggests such cases can hardly pass through additional sampling alone.

To directly compare the effectiveness of preference pairs generated via iterative debugging versus sampling, we conducted experiments using 10k training samples. Table 7 presents these comparative results. As shown in Table 7, iterative debugging can generate more meaningful preference pairs than sampling by leveraging interpreter feedback and runtime information, thereby achieving better results (e.g., an average score of 64.1 vs 62.3) with lower API costs.

A.2 Synthetic Test Case Generation

A.2.1 Rationale for using Synthetic Test Cases

We address the relationale behind systhetic test cases from the following perpectives. Optimizing code LLMs through preference learning requires a large amount of training data, which is difficult to annotate or verify manually. Synthetic data has become a widely adopted approach. For example, Qwen2.5-Coder utilizes tens of millions of synthetic instruction samples, and models like DeepSeek-V3 and R1 also incorporate synthetic data during training, also as demonstrated in studies like PLUM, SelfCodeAlign and DSTC.

A.2.2 Validity of Synthetic Test Cases

We have made the following efforts to ensure the qaulity of test cases: (i) First, we adopted a powerful LLM, GPT-4o, as the test case generator to primarily ensure its validity. (ii) Through prompting engineering, we have invested significant effort into making the generated test cases broad and meaningful. (iii) We conducted a manual evaluation by performing a random sample check. We manually examined 100 data samples and found that all the generated test cases correctly reflected the task requirements. However, we observed that these test cases tend to be relatively simple and may not cover all edge cases.

A.2.3 Coverage Analysis of Test Cases

To validate the effectiveness of test cases in exercising source code, we conducted coverage analysis on a sample of 1,000 training instances. Code coverage, a crucial metric in software testing, quantifies the extent to which a program's source code is exercised by test cases. This metric measures

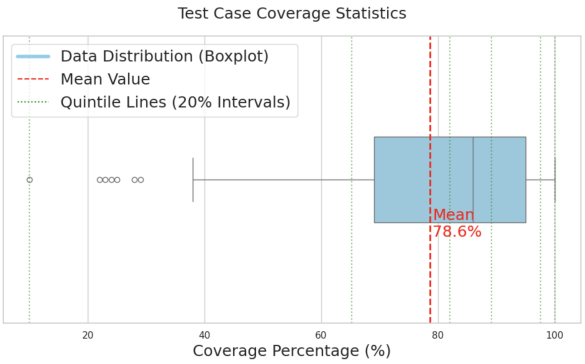

Figure 7: Coverage Distribution of Test Case.

the percentage of code executed by a test suite, which we evaluated using the Python Coverage library[1]. Our evaluation framework organizes the source code and test cases into directories, maintaining a clear separation between the source code (implemented as Python modules) and the corresponding unit tests (developed using the unittest[2] framework). The coverage metric is calculated through the following equation:

$$\text{Code Coverage} = 100\% \times \left(\frac{\text{Number of lines of code executed}}{\text{Total Number of lines of code in system component}} \right). \tag{9}$$

The results of the coverage analysis are shown in Figure 7.

A.2.4 Evaluating the quality of test cases using LLM-as-a-judge

To evaluate test case quality, we employ the LLM-as-a-judge approach, assessing three dimensions: **accuracy**, **effectiveness**, and **reasonableness** on a 5-point scale. Detailed evaluation prompts are provided in 9. We sample 1,000 data points from the training data and conduct evaluations using the `DeepSeek-V3-0324` model. The evaluation results in Figure 8 demonstrate satisfactory test case quality across all dimensions.

A.2.5 Prompt and Examples for Generated Test Cases

To illustrate our synthetic test case generation process, Figure 10 displays the prompt template provided to the LLM. Following this prompt, Figure 11 presents an example of the test cases generated

[1] https://github.com/nedbat/coveragepy
[2] https://docs.python.org/3/library/unittest.html

	HumanEval	HumanEval+	MBPP	MBPP+	BCB-Inst	BCB-Comp	LCB-v5	Avg.
Ref.	61.6	53.0	76.9	62.9	40.2	45.8	24.1	52.1
Sampling	84.1	79.3	82.3	68.5	40.8	49.2	32.4	62.3
Debugging	**87.8**	**84.8**	**83.3**	**69.7**	**41.1**	**49.6**	**32.5**	**64.1**

Table 7: Results of Preference Pair Construction using Sampling vs. Debugging (10k training samples).

Figure 8: Comparative distribution of LLM-as-a-judge scores across three evaluation metrics: accuracy, effectiveness, and reasonableness.

for question "Create a Python function named 'generate_arithmetic_sequence' that generates the first N terms of an arithmetic sequence given the first term and the common difference. The function should take three parameters: the first term (a), the common difference (d), and the number of terms (N). The function should return a list containing the first N terms of the sequence".

A.3 Training and Inference Parameters

Unless specified otherwise for a particular experiment, our code LLMs were trained with consistent hyperparameter settings. The learning rate was set to 1×10^{-5} for the 7B code LLMs and 5×10^{-6} for the 14B models. We used a global batch size of 128, full-parameter training 3 epochs with max prompt length of 1024 and generation length of 2048. A cosine learning rate scheduler with was employed, with 3% of the total training steps dedicated to warm-up. For the DPO algorithm, β is set to 0.1, and α is set to 1.0 for RPO. π_θ and π_{ref} are both initialized with the weights of the evaluated model, while π_{ref} keeps frozen during training. For all inference, greedy decoding was utilized with pass rate at first attempt reported. For Code-Optimise, we replicated their setup, and the DPO_{PvF} setting results are reported. For PLUM and CodeDPO, their scores are taken from their respective papers.

B Experimental Results and Comparisons

This section begins by describing the evaluation benchmarks and their statistics. It then presents the comprehensive experimental results of our method on these benchmarks, followed by detailed comparisons against Supervised Fine-Tuning (SFT) and other relevant state-of-the-art methods.

B.1 Evaluation Benchmarks: Description and Statistics

We detail the individual function-level code generation benchmarks used for evaluation in this subsection. Table 8 summarizes key statistics for these benchmarks, such as the number of problems and the average number of tests per problem.

Dataset	Problems	Avg. Tests
HumanEval	164	9.57
HumanEval+		748.07
MBPP	378	3.11
MBPP+		105.40
LiveCodeBench	Easy 279	18.07
	Medium 331	21.81
	Hard 270	24.78

Table 8: Statistics of Evaluation Benchmarks.

HumanEval and MBPP are popular benchmarks for assessing code generation. Considering the limited test cases in these benchmarks (HumanEval: 9.57 avg. tests; MBPP: 3.11 avg. tests, as seen in Table 8), we followed previous work and utilized the EvalPlus framework to evaluate model robustness across a broader range of test cases (HumanEval+: 748.07 avg. tests; MBPP+: 105.40 avg. tests). To ensure fair comparison, we used version 0.2.0 of MBPP+ provided by EvalPlus[3] v0.3.1 , which removes some broken tasks (399 $\rightarrow$ 378 tasks).

BigCodeBench (BCB) is a comprehensive benchmark designed to assess a model's ability

[3] https://github.com/evalplus/evalplus

````
You are an expert software quality analyst. Evaluate test cases based on the provided code an
d test code.

Input Format
```json
{
  "code": "[target code snippet]",
  "test_code": "[test case implementation]"
}
```

Evaluation Criteria (Score 1-5 per dimension)
###1 Accuracy
• 5: Tests cover all functional requirements & edge cases
• 3: Partial coverage with missed edge scenarios
• 1: Tests fail basic functionality validation
###2 Effectiveness
• 5: Tests detect >90% potential defects via mutation testing
• 3: Detects obvious errors but misses logical flaws
• 1: Tests pass even with broken implementations
###3 Reasonableness
• 5: Logical assertions, minimal redundancy
• 3: Some overlapping/duplicate tests
• 1: Contradictory or irrational test logic

Evaluation Protocol
1. Requirements Mapping
   Cross-reference tests with code functionality and context docs
2. Boundary Analysis
   Check if tests validate:
   • Input extremes (min/max values)
   • Exception handling paths
   • State transitions
3. Mutation Resistance Test
   Mentally simulate common code mutations (e.g., change operators, remove null checks). Ver
ify if tests catch these.

Output Format
```json
{
  "scores": {
    "accuracy": "[1-5]",
    "effectiveness": "[1-5]",
    "reasonableness": "[1-5]"
  }
}
```
Example Response
```json
{
  "scores": {
    "accuracy": 4,
    "effectiveness": 3,
    "reasonableness": 5
  }
}
```
````

Figure 9: Prompt for evaluating the quality of test cases using LLM-as-a-judge.

Prompt for Generating Code and Test Cases

Now that you are a code expert, I have provided you with the QUESTION. Complete the problem with awesome code logic and give a richly commented analysis in the code of your answer. Include the necessary packages and test cases.

- QUESTION

{task}

- Full code implementation with test cases

Enclose the python code with ```python and ``` and enclose the file name with <file> and </file>. For example:

<file>add.py</file>

```python
# add.py
# Code implementation here
def add(x, y):
return x + y
```

The test code should be in a single file.

<file>test.py</file>

Note that the following code will be executed directly, so only the test cases that can be executed directly need to be retained. You only need to test some simple functions in the code. Tests that depend on external files cannot be executed because these files do not exist.

```python
from add import add
def test():
assert add(3, 5) == 8
assert add(4, 6) == 10
test()
```

- File names in order and packages required

Answer file names and packages in JSON format, wrapped in <json> and </json> tags. For example:

<json>
{
"file_names": ["add.py", "test.py"],
"packages": ["package1", "package2"]
}</json>

Figure 10: Prompt used for generating code and test cases.

to handle real-world programming tasks, particularly its effectiveness in utilizing various function calls as tools. Our model's ability to adeptly manage these high-complexity scenarios underscores its suitability for BigCodeBench.

LiveCodeBench (LCB), statistics for which are also included in Table 8 (showing problem distribution by difficulty), is a benchmark designed to evaluate code generation models on challenging competitive programming problems, often sourced from real coding contests. Unlike benchmarks focused solely on function completion, LCB tasks typically require more complex algorithmic reasoning and problem-solving skills. The evaluation often simulates a contest environment, potentially including hidden test cases to assess the robustness and correctness of the generated solutions under pressure. In our experiments, we utilize LiveCodeBench-v5 (LCB-v5) to gauge the model's capabilities in tackling these demanding, contest-style coding scenarios.

B.2 Comparison with Supervised Fine-Tuning

We compare our method with standard Supervised Fine-Tuning (SFT) to demonstrate the benefits of our preference alignment approach. The overall comparison of our method (Target-RPO) against SFT and reference models on various benchmarks is presented in Table 9. For instance, on Qwen2.5-Coder-7B, our Target-RPO achieves an average of 65.3, surpassing the SFT baseline's 63.2.

Table 10 provides a more granular breakdown of performance on LiveCodeBench-v5 by difficulty, where our method consistently outperforms the SFT versions of both CodeQwen-7B-Chat and Qwen-Coder-7B, especially on Easy and Medium problems.

Similarly, Table 11 shows detailed results on BigCodeBench categories, again illustrating the advantage of our approach over SFT across different task types and difficulties. For example, Qwen-Coder-7B with our method achieves an average of 36.8 compared to SFT's 31.8.

B.3 Comparison with Other Code Generation Methods

We further benchmark our method against other notable code generation techniques. Table 12 benchmarks our method against techniques like CodeDPO, Code-Optimise, and PLUM on HumanEval

Generated Test Cases

```
import unittest
from generate_arithmetic_sequence import generate_arithmetic_sequence

class TestGenerateArithmeticSequence(unittest.TestCase):

    def test_basic_case(self):
        """Test a standard arithmetic sequence with positive common difference."""
        self.assertEqual(generate_arithmetic_sequence(2, 3, 5), [2, 5, 8, 11, 14], "Basic case failed")

    def test_single_term(self):
        """Test when only one term (N=1) is requested."""
        self.assertEqual(generate_arithmetic_sequence(5, 10, 1), [5], "Single term case failed")

    def test_zero_terms(self):
        """Test when zero terms (N=0) are requested; should return an empty list."""
        self.assertEqual(generate_arithmetic_sequence(7, 3, 0), [], "Zero terms case failed")

    def test_negative_number_of_terms(self):
        """Test when a negative number of terms (N<0) is requested; should return an empty list."""
        self.assertEqual(generate_arithmetic_sequence(5, 3, -1), [], "Negative N (-1) case failed")
        self.assertEqual(generate_arithmetic_sequence(5, 3, -5), [], "Negative N (-5) case failed")

    def test_negative_common_difference(self):
        """Test a sequence with a negative common difference (decreasing sequence)."""
        self.assertEqual(generate_arithmetic_sequence(10, -2, 4), [10, 8, 6, 4], "Negative common difference case
failed")

    def test_zero_common_difference(self):
        """Test a sequence where all terms are the same (common difference is zero)."""
        self.assertEqual(generate_arithmetic_sequence(5, 0, 4), [5, 5, 5, 5], "Zero common difference case failed")
        self.assertEqual(generate_arithmetic_sequence(-3, 0, 3), [-3, -3, -3], "Zero common difference with negative
start failed")

    def test_mixed_sign_inputs(self):
        """Test with mixed positive and negative inputs for 'a' and 'd' where sequence crosses zero."""
        self.assertEqual(generate_arithmetic_sequence(-5, 2, 5), [-5, -3, -1, 1, 3], "Mixed sign inputs case failed")

    def test_float_inputs(self):
        """Test with floating-point numbers for the first term and common difference."""
        self.assertEqual(generate_arithmetic_sequence(1.5, 0.5, 4), [1.5, 2.0, 2.5, 3.0], "Float inputs case failed")
        self.assertEqual(generate_arithmetic_sequence(0.0, -0.1, 3), [0.0, -0.1, -0.2], "Float inputs with negative
difference failed")
        # Example where direct float comparison could be tricky, but Python handles these well.
        self.assertEqual(generate_arithmetic_sequence(0.1, 0.1, 3), [0.1, 0.2, 0.3], "Float inputs (0.1 increments)
failed")

    def test_large_number_of_terms(self):
        """Test with a large number of terms for basic performance and correctness of first/last terms."""
        N_large = 1000
        a_val = 1
        d_val = 1
        sequence = generate_arithmetic_sequence(a_val, d_val, N_large)

        self.assertEqual(len(sequence), N_large, "Large N: Length mismatch")
        if N_large > 0:
            self.assertEqual(sequence[0], a_val, "Large N: First term mismatch")
            # The Nth term (index N-1) is a + (N-1)*d
            self.assertEqual(sequence[-1], a_val + (N_large - 1) * d_val, "Large N: Last term mismatch")

    def test_invalid_N_type(self):
        """Test with non-integer N; should return an empty list."""
        self.assertEqual(generate_arithmetic_sequence(1, 1, 3.5), [], "Non-integer N (float) case failed")
        self.assertEqual(generate_arithmetic_sequence(1, 1, "abc"), [], "Non-integer N (string) case failed")

if __name__ == '__main__':
    # Running the tests
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Figure 11: An example of generated test cases.

	HumanEval		MBPP		BCB-Inst	BCB-Comp	LCB-v5	Average
	Base	*Plus*	*Base*	*Plus*				
				CodeQwen1.5-7B-Chat				
Ref.	83.5	78.7	79.4	69.0	39.6	43.6	15.3	58.4
SFT	87.8	83.5	82.3	69.6	35.9	45.6	17.0	60.2
Target-DPO	89.6	85.4	**83.9**	69.8	**39.9**	**48.7**	**20.2**	**62.5**
Target-RPO	**89.6**	**86.0**	82.5	**70.4**	38.3	48.4	19.9	62.2
				Qwen2.5-Coder-7B				
Ref.	61.6	53.0	76.9	62.9	40.2	45.8	24.1	52.1
SFT	87.2	82.9	83.1	68.3	39.1	51.6	30.0	63.2
DiffAug-RPO	86.0	81.7	82.8	67.5	40.7	51.4	30.5	62.9
Target-RPO	**89.6**	**84.8**	**83.3**	**69.5**	**43.1**	**53.3**	**33.3**	**65.3**

Table 9: SFT and our results on CodeQwen1.5-7B-Chat and Qwen2.5-Coder-7B. Detailed results on LiveCodeBench and BigCodeBench are presented in Table 10 and Table 11.

	EASY	MEDIUM	HARD	Avg
CodeQwen-7B-Chat-SFT	41.87	9.14	0.75	17.06
CodeQwen-7B-Chat-Ours	**48.73**	**11.89**	**0.75**	**20.16**
Qwen-Coder-7B-SFT	67.14	21.03	2.61	30.01
Qwen-Coder-7B-Ours	**69.31**	**27.13**	**3.73**	**33.33**

Table 10: Detailed Results on LiveCodeBench-v5, comparing SFT with Our Target-DPO.

	Complete-Full	Instruct-Full	Complete-Hard	Instruct-Hard	Average
CodeQwen-7B-Chat-SFT	45.6	35.9	18.3	15.5	28.8
CodeQwen-7B-Chat-Ours	**48.4**	**38.3**	**20.3**	**18.2**	**31.3**
Qwen-Coder-7B-SFT	51.6	39.1	21.6	14.9	31.8
Qwen-Coder-7B-Ours	**53.3**	**43.1**	**29.7**	**20.9**	**36.8**

Table 11: Detailed Results on BigCodeBench, comparing SFT with Our Target-DPO.

and MBPP. Notably, on the DeepSeekCoder-6.7B base, our method achieves significantly higher scores (e.g., 66.50 on HumanEval vs. 59.75 for CodeDPO and 56.70 for PLUM).

C In-depth Analyses and Ablation Studies

In this section, we conduct several in-depth analyses and ablation studies to better understand the characteristics and behavior of our proposed method. This includes investigating the impact of model size (scaling laws), sensitivity to key hyperparameters (β and α), the diversity of our constructed preference data, common error patterns in the generated code, and the efficiency of our preference annotation process.

C.1 Scaling Law on Model Size

The impact of model size on performance when applying our method is detailed in Table 13. The results show a clear trend: as model size increases from 1.5B to 32B parameters, the average performance improves from 54.4 to 74.7, demonstrating the scalability of our approach.

C.2 Ablation Studies on Hyperparameters β and α

In Direct Preference Optimization (DPO), the hyperparameter β controls the strength of the preference signal, essentially determining how strictly the model should adhere to the learned preferences relative to the reference model. The hyperparameter α, when part of the DPO framework or a combined loss, often serves as a weighting factor for an additional objective or regularization term. We performed ablation studies on key hyperparameters β and α. Table 14 presents the results for β when α is set to 0, suggesting that a smaller β (e.g., 0.1) yields the best average performance (69.7).

The corresponding ablation for α, with β fixed at 0.1, is shown in Table 15. These results indicate that $\alpha = 1.0$ provides the highest average score (70.5), while $\alpha = \infty$ (equivalent to SFT) performs relatively worse.

C.3 Ablation on Context Usage

Beyond CodeQwen1.5-7B-Chat, we also provide ablation results on the use of context from the rejected sample using Qwen2.5-Coder-7B, as shown in Table 16.

C.4 Analysis of Preference Data Diversity

To understand the characteristics of our CodeFlow preference dataset, Table 17 provides a distributional analysis of various features (e.g., Workflow, Functionality, Data Processing) across 1k samples, comparing it with other common datasets like Alpaca and OSS-Instruct. Our CodeFlow dataset (both preferred and dis-preferred samples) generally exhibits a higher count and thus potentially greater diversity across most features, particularly in Data Processing, File Operation, and Advanced Techniques.

C.5 Error Analysis of Generated Code

Target-DPO Generates Fewer Errors. In this section, we present a statistical analysis of common failure case types to pinpoint frequent pitfalls in code generation. By contrasting critical tokens between a corrected version and its preceding iteration explicitly, a Code LLM equipped with Target-DPO makes fewer errors. Table 18 presents the frequency of common failure types (e.g., AttributeError, KeyError) on the BigCodeBench Complete-Full set. Our Target-RPO method shows a notable reduction in the sum of these errors (308 occurrences) compared to RPO (369) and Code-Optimise-RPO (396) on Qwen2.5-Coder-7B. This suggests that while RPO includes SFT, it still requires targeted learning of critical errors in the dis-preferred samples to effectively reduce mistakes.

C.6 Efficiency Analysis of Preference Annotation

Additionally, we compare the costs of generating and annotating preference pairs to guide more efficient preference alignment and reduce errors.

Target-DPO Provides an Efficient Pathway for Preference Annotation. We compare the cost of synthesizing one preference pair between Target-DPO and the sampling techniques adopted by Code-Optimise, primarily considering external LLM calls and execution times. Given an instruction, Code-Optimise synthesizes m code snippet candidates (where m is often set to 100), using n test cases from the raw dataset, leading to $m \times n$ executions on the CPU. In contrast, for a single instruction, Target-DPO requires up to 7 LLM calls and executions for successful pair generation in most cases. Considering the failure ratio (when code can't pass the generated test cases within the

	HumanEval	HumanEval+	MBPP	MBPP+
StarCoder2-7B	35.40	29.90	54.40	45.60
CodeDPO	48.17	34.15	58.40	49.37
Code-Optimise	32.32	28.05	58.90	47.89
PLUM	46.30	39.60	60.40	49.10
Our Target-DPO	**48.20**	**43.90**	**63.50**	**50.60**
DeepSeekCoder-1.3B	31.53	28.65	57.40	48.67
CodeDPO	42.07	38.04	61.37	53.43
Code-Optimise	34.15	30.49	59.15	49.87
Our Target-DPO	**47.00**	**43.30**	**61.37**	**54.20**
DeepSeekCoder-6.7B	47.60	39.60	70.20	56.60
CodeDPO	59.75	51.83	72.18	60.01
Code-Optimise	47.56	37.20	72.18	57.64
PLUM	56.70	48.80	72.90	58.90
Our Target-DPO	**66.50**	**60.40**	**76.50**	**61.40**

Table 12: Performance comparison with baselines.

Model	HumanEval		MBPP		BCB-Inst	BCB-Comp	Average
	Base	*Plus*	*Base*	*Plus*			
Qwen2.5-Coder-1.5B	67.7	62.8	66.7	55.3	33.4	40.5	54.4
Qwen2.5-Coder-7B	89.6	84.8	83.3	69.5	43.1	53.3	70.6
Qwen2.5-Coder-32B	**92.7**	**86.6**	**89.4**	**74.6**	**45.7**	**58.9**	**74.7**

Table 13: Ablations on model size.

β	HumanEval		MBPP		BCB-Inst	BCB-Comp	Average
	Base	*Plus*	*Base*	*Plus*			
0.1	**89.0**	**83.6**	83.1	69.0	**41.0**	**52.7**	**69.7**
0.3	86.6	80.5	82.5	**68.5**	40.3	50.3	69.1
0.5	85.4	80.5	**83.6**	66.7	40.5	48.2	67.5

Table 14: Ablations on β with α set 0.

α	HumanEval		MBPP		BCB-Inst	BCB-Comp	Average
	Base	*Plus*	*Base*	*Plus*			
1.0	**89.6**	**84.8**	**83.1**	69.3	**43.1**	53.3	**70.5**
3.0	87.2	81.1	82.8	69.0	40.3	52.7	67.9
5.0	84.1	80.5	83.1	**70.1**	40.3	**54.0**	68.7
∞ (SFT)	87.2	82.9	83.1	68.3	39.1	51.6	68.7

Table 15: Ablations on α with β set 0.1.

	HumanEval		MBPP		BCB-Inst	BCB-Comp	LCB-v5	Average
	Base	*Plus*	*Base*	*Plus*				
Qwen2.5-Coder-7B	61.6	53.0	76.9	62.9	40.2	45.8	24.1	52.1
SFT	87.2	82.9	83.1	68.3	39.1	51.6	30.0	63.2
Hybrid-RPO	82.9	79.3	81.7	67.5	41.2	50.5	29.8	61.8
DiffAug-RPO	86.0	81.7	82.8	67.5	40.7	51.4	30.5	62.9
Target-RPO	**89.6**	**84.8**	**83.3**	**69.5**	**43.1**	**53.3**	**33.3**	**65.3**

Table 16: Ablation results on the Target-DPO using Qwen2.5-Coder-7B.

Datasets	Workflow	Functionality	Computation Operation	User Interaction	Data Processing	File Operation
Alpaca	994	393	282	82	221	11
CodeFeedback	2079	535	689	143	895	39
Evol-Alpaca	2163	591	783	134	1401	55
OSS-Instruct	2254	669	413	192	903	102
CodeFlow (Preferred)	**2689**	**805**	**967**	**410**	**2418**	**290**
CodeFlow (Dis-Preferred)	2490	772	964	406	2327	287

	Logging	Algorithm	Data Structures	Implementation Logic	Advanced Techniques	Average
Alpaca	1	232	72	67	10	215.00
CodeFeedback	10	427	100	49	63	457.18
Evol-Alpaca	15	414	130	74	94	532.18
OSS-Instruct	62	150	140	82	26	453.91
CodeFlow (Preferred)	**133**	**790**	**367**	**152**	**178**	**836.27**
CodeFlow (Dis-Preferred)	129	785	361	149	193	805.73

Table 17: Distribution of total features across 1k samples.

Error Type	RPO	Code-Optimise-RPO	Target-RPO (Ours)
AttributeError: 'X' has no attribute 'Y'	145	149	127
KeyError: 'X'	139	128	100
NameError: name 'X' is not defined	41	58	46
FileNotFoundError: No such file or directory	44	61	35
Sum	369	396	**308**

Table 18: The frequency of the most common failure types on the BigCodeBench Complete-Full set.

budget), an estimated 10.4 calls are needed for a given instruction on average across the dataset. This is far fewer than the m (e.g., 100) calls for sampling candidates plus subsequent executions often employed by sampling-heavy methods. Though massive sampling can yield diverse candidates, it is not efficient as most code snippets are discarded. Target-DPO shows that starting with a single code snippet, even if it fails initially, it still holds high potential to form a valuable preference pair for alignment training through iterative refinement.

C.7 Alternatives to LCS for Code Differencing

To explore alternatives beyond LCS-based code differencing, we considered a simple baseline approach where we extract the common prefix and suffix, treating the middle part as the difference. The results are shown in Table 19:

Consider these two versions of a Python function where only two lines are truly different:

Listing 1: Chosen Code

```
def hello():
    print("Hello")    # <-- difference
    x = 1             # <-- unchanged
    y = 2             # <-- unchanged
    return x + y      # <-- difference

z = hello()
```

Listing 2: Rejected Code

```
def hello():
    print("Hi")       # <-- difference
    x = 1             # <-- unchanged
    y = 2             # <-- unchanged
    return x * y      # <-- difference

z = hello()
```

Method	HE	HE+	MBPP	MBPP+	BCB-Comp	BCB-Inst	LCB-v5	Avg.
Ref.	61.6	53.0	76.9	62.9	45.8	40.2	24.1	52.1
Prefix-Suffix	86.6	79.9	81.5	68.0	47.8	39.2	30.1	61.9
LCS	**89.0**	**83.6**	**83.1**	**69.0**	**52.7**	**41.0**	**32.6**	**64.4**

Table 19: Comparison of Prefix-Suffix and LCS methods for code difference extraction across benchmarks.

Method	HE	HE+	MBPP	MBPP+	BCB-Comp	BCB-Inst	LCB-v5	Avg.
Ref.	61.6	53.0	76.9	62.9	45.8	40.2	24.1	52.1
First	87.8	81.7	81.0	67.5	47.2	40.4	29.7	62.2
Last	88.4	82.9	82.0	68.0	51.7	41.2	31.5	63.7
Random (Ours)	**89.0**	**83.6**	**83.1**	**69.0**	**52.7**	**41.0**	**32.6**	**64.4**

Table 20: Comparison of different negative sample selection strategies across benchmarks.

Method	HE	HE+	MBPP	MBPP+	BCB-Comp	BCB-Inst	LCB-v5	Avg.
Ref.	61.6	53.0	76.9	62.9	45.8	40.2	24.1	52.1
DS-v3-SFT	82.3	76.2	83.3	69.8	46.9	40.7	26.4	60.8
DS-v3-Ours	**84.8**	**79.3**	**84.7**	**72.0**	**47.5**	**40.7**	**27.7**	**62.4**

Table 21: Results of distillation from DeepSeek-V3 to Qwen2.5-Coder-7B.

Prefix-Suffix-based method incorrectly marks both `x = 1` and `y = 2` as differences, while the LCS-based method correctly identifies only the two truly changed lines (`print("Hi")` and `return x * y`, providing a more precise grounding of the differences, enabling more targeted training.

C.8 Different Negative Samples

We chose the random negative selection method to better simulate the variety of error types in real-world scenarios, where some versions are severely flawed and others only slightly. This random slection approach enhances the diversity of negative samples and helps improve generalization.

To validate this choice, we conduct additional experiments using only the first or last incorrect versions as negatives. The results in Table 20 show that our method consistently achieves superior performance on six benchmarks.

C.9 Different Teacher Model

We adopt DeepSeek-V3 (671B MoE LLM with 37B active parameters) as the teacher model and synthesize a total of 28k pairs with resource constraints. The results using Qwen2.5-Coder-7B as the student model are shown in Table 21. Target-DPO demonstrates generalizable improvement when using different teachers for distillation.