Aligning with Human Coding Preferences for Improving Code Generation

XIN YIN, Zhejiang University, China
CHAO NI*, Zhejiang University, China
XIAOHU YANG, Zhejiang University, China

Large Language Models (LLMs) have demonstrated remarkable potential in automating software development tasks. While recent advances leverage Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO) to align models with human preferences, the optimal training strategy remains unclear across diverse code preference types. This paper systematically investigates the roles of SFT and DPO in aligning LLMs with different code preferences. Through both theoretical analysis and empirical observation, we hypothesize that SFT excels in types with objectively verifiable optimal solutions, while applying SFT followed by DPO (S&D) enables models to explore superior solutions in types without objectively verifiable optimal solutions. Based on the analysis and experimental evidence, we propose **A**daptive **P**reference **O**ptimization (APO), a dynamic integration approach that adaptively amplifies preferred responses, suppresses dispreferred ones, and encourages exploration of potentially superior solutions during training. Extensive experiments across six representative code preference tasks validate our theoretical hypotheses and demonstrate that APO consistently matches or surpasses the performance of existing SFT and S&D strategies. Our work provides both theoretical foundations and practical guidance for selecting appropriate training strategies in different code preference alignment types.

CCS Concepts: • **Software and its engineering** → Software maintenance tools.

Additional Key Words and Phrases: Code Generation, Large Language Model, Human Coding Preference

ACM Reference Format:
Xin Yin, Chao Ni, and Xiaohu Yang. 2026. Aligning with Human Coding Preferences for Improving Code Generation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE106 (July 2026), 23 pages. https://doi.org/10.1145/3808113

1 Introduction

In recent years, Large Language Models (LLMs) have emerged as transformative tools with remarkable potential for automating diverse software development tasks. State-of-the-art models such as CodeLlama [34], WizardCoder [21], and DeepSeek-Coder [13] have demonstrated impressive capabilities in tackling complex code generation tasks. These models excel at generating code snippets from natural language descriptions [29, 53], performing cross-language code translation [12, 25], automatically fixing software bugs [48, 50], and synthesizing comprehensive unit tests to enhance software reliability [20, 51]. Despite these advances, a critical challenge remains in effectively aligning LLMs with human code preferences (e.g., the security and efficiency of the code).

*This is the corresponding author. Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, Hangzhou, China.

Authors' Contact Information: Xin Yin, Innovation and Management Center (Ningbo) for School of Software Technology, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, xyin@zju.edu.cn; Chao Ni, Innovation and Management Center (Ningbo) for School of Software Technology, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, chaoni@zju.edu.cn; Xiaohu Yang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, yangxh@zju.edu.cn.

ACM 2994-970X/2026/7-ARTFSE106
https://doi.org/10.1145/3808113

To improve code generation models, a common approach is Supervised Fine-Tuning (SFT) [55], where models are trained on pairs of instructions and corresponding correct code snippets. Current methods generate synthetic training data through self-instruction using models like GPT-4 [4, 40, 44]. Evol-Instruct [21] leverages more sophisticated prompts for enhanced data generation, while OSS-Instruct [45] enables LLMs to draw inspiration from real-world code snippets to improve coding performance. However, SFT's exclusive focus on correct examples limits its ability to teach preference discrimination, as models never encounter negative examples [15].

Recent research addresses these limitations through Direct Preference Optimization (DPO) [32], which aligns models using pairwise preference data. DPO enables models to rank outputs and select preferred solutions (e.g., more efficient code) [30, 46, 53, 54]. Prior works [24, 31, 33, 39] reveal distinct learning behaviors between SFT and DPO, as illustrated in Fig. 1. Specifically, confidence scores for both $\mathbf{y}^+$ and $\mathbf{y}^-$ gradually decrease during DPO training, while $\mathbf{y}^+$ confidence remains stable under SFT (detailed in Section 2.2). Although SFT and DPO have shown success in natural language tasks, their effectiveness across diverse code tasks remains under-explored. Code preferences differ fundamentally from natural language, requiring objective metrics such as correctness, efficiency, and security. Given their distinct behaviors, SFT and DPO may serve different roles depending on the specific preference type, complicating the selection of optimal training strategies.

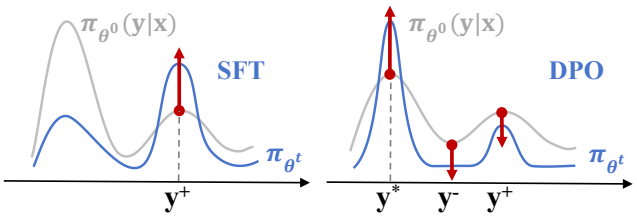

Fig. 1. The updated confidence scores of different training algorithms. The x-axis represents different responses $\mathbf{y}$, while the y-axis indicates the model's confidence score for each response. Here, $\mathbf{y}^+$ and $\mathbf{y}^-$ denote the preferred and dispreferred responses, respectively, and $\mathbf{y}^*$ represents the response with the highest confidence before training. Gray and blue markers correspond to confidence scores before and after training, respectively, while red arrows illustrate the post-training trends in confidence scores for $\mathbf{y}^+$, $\mathbf{y}^-$, and $\mathbf{y}^*$.

In this paper, we first conduct a preliminary experiment to verify the reported phenomena [24, 31, 33, 39] and provide a theoretical analysis to explain these behaviors. We categorize code preference tasks into two types: ❶ preferences with objectively verifiable optimal solutions, and ❷ preferences without objectively verifiable optimal solutions. Based on these observations and theoretical analysis (detailed in Section 3), we formulate the following hypotheses regarding SFT and DPO effectiveness. In type ❶, SFT suffices to achieve optimal solutions, while DPO provides limited additional benefits and may even harm performance. Conversely, in type ❷, sequential application of SFT followed by DPO (S&D) achieves optimal performance, where SFT rapidly builds fundamental capabilities and DPO subsequently enables exploration of superior solutions.

Building on the analysis and experimental evidence (detailed in Section 3 and Section 5), we propose **A**daptive **P**reference **O**ptimization (APO), a unified framework that dynamically integrates the respective strengths of SFT and DPO. APO amplifies preferred responses, suppresses dispreferred ones, and encourages exploration of superior solutions during training.

To validate these hypotheses and evaluate APO's effectiveness, we construct six code preference tasks using the APPS benchmark [14]. Type ❶ encompasses code correctness, security, and smell optimization, while type ❷ covers efficiency, complexity, and conciseness optimization. Experimental results validate our hypotheses: SFT achieves superior stability in type ❶, while (S&D) delivers

optimal performance in type ❷. Beyond validating these hypotheses, APO demonstrates additional practical benefits. Compared with the standard SFT+DPO strategy, APO not only simplifies the training process (requiring just a single deployment and 5 training epochs), but also improves output quality. Overall, APO enlarges the model's exploration space while increasing the probability of positive samples and decreasing the probability of negative samples. In the later training stages, the model keeps exploring better solutions (e.g., more efficient code). Another key benefit of APO is that it reduces the need for human intervention and avoids misusing training strategies. Humans no longer need to determine the task type and select different strategies accordingly. Instead, they can simply adopt APO to achieve near-optimal performance on type ❶ (on par with the best SFT) and superior performance on type ❷ (outperforming the best SFT+DPO). We also compare the efficiency of the proposed APO framework with SFT and DPO. APO also maintains competitive efficiency in training time and GPU memory usage.

Our main contributions are summarized as follows:

- We theoretically analyze SFT and DPO behaviors and propose type-specific hypotheses for code preference alignment, providing guidance for training strategy selection.
- We introduce **A**daptive **P**reference **O**ptimization (APO), a unified framework that adaptively combines SFT and DPO strengths without requiring manual type discrimination.
- We empirically validate our hypotheses across six code preference tasks and demonstrate APO's superior performance while maintaining competitive training efficiency.

2 Preliminary Study

2.1 Background

Code Preference. The term "preference" originates from preference optimization techniques and is widely used in previous studies [32, 53, 54] to capture human tendencies toward certain desirable properties. In the context of code generation, we define *code preferences* as the human tendencies toward specific characteristics of code (e.g., correctness and efficiency).

Language Model. We formalize an LLM as a conditional policy $\pi_\theta(\mathbf{y} \mid \mathbf{x})$ parameterized by θ, mapping user instructions $\mathbf{x} \in \mathcal{X}$ to textual responses $\mathbf{y} \in \mathcal{Y}$. Given input $\mathbf{x}$, the LLM generates response $\mathbf{y}$ auto-regressively:

$$\pi_\theta(\mathbf{y} \mid \mathbf{x}) = \prod_t \pi_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{<t}), \tag{1}$$

where $\mathbf{y}_t$ and $\mathbf{y}_{<t}$ denote the t-th token and its preceding tokens, respectively.

SFT. Supervised Fine-Tuning (SFT) [55] trains models to generate high-quality responses by learning from demonstration data. Intuitively, SFT teaches the model "what good responses look like" by maximizing the likelihood of preferred responses $\mathbf{y}^+$ given prompts $\mathbf{x}$. This process constructs a dataset of instruction-response pairs $\mathcal{D} = \{(\mathbf{x}, \mathbf{y}^+)\}$ and optimizes π_θ via the SFT loss:

$$\mathcal{L}_{\text{SFT}}(\pi_\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y}^+)\sim\mathcal{D}}\left[\log \pi_\theta(\mathbf{y}^+ \mid \mathbf{x})\right]. \tag{2}$$

DPO. Direct Preference Optimization (DPO) [32] represents a key advancement in LLM alignment, adapting models to better reflect human preferences. Unlike SFT, which only demonstrates desirable behaviors, DPO explicitly discourages undesirable responses through preference learning. DPO directly optimizes policy π_θ using preference data constructed as triples $\mathcal{D} = \{(\mathbf{x}, \mathbf{y}^+, \mathbf{y}^-)\}$, where $\mathbf{y}^+$ and $\mathbf{y}^-$ represent preferred and dispreferred responses to prompt $\mathbf{x}$, respectively. Given a prompt $\mathbf{x}$, DPO increases the relative likelihood of preferred response $\mathbf{y}^+$ compared to dispreferred response

$\mathbf{y}^-$. The DPO loss is defined as:

$$\begin{aligned}\mathcal{L}_{\text{DPO}}(\pi_\theta) &= -\mathbb{E}_{(\mathbf{x},\mathbf{y}^+,\mathbf{y}^-)\sim\mathcal{D}}\left[\log\sigma\left(\beta\left(\log\frac{\pi_\theta(\mathbf{y}^+\mid\mathbf{x})}{\pi_{\text{ref}}(\mathbf{y}^+\mid\mathbf{x})}-\log\frac{\pi_\theta(\mathbf{y}^-\mid\mathbf{x})}{\pi_{\text{ref}}(\mathbf{y}^-\mid\mathbf{x})}\right)\right)\right]\\ &= \mathbb{E}_{(\mathbf{x},\mathbf{y}^+,\mathbf{y}^-)\sim\mathcal{D}}\left[\log\left(1+\left(\frac{\pi_{\text{ref}}(\mathbf{y}^+\mid\mathbf{x})\cdot\pi_\theta(\mathbf{y}^-\mid\mathbf{x})}{\pi_{\text{ref}}(\mathbf{y}^-\mid\mathbf{x})\cdot\pi_\theta(\mathbf{y}^+\mid\mathbf{x})}\right)^\beta\right)\right].\end{aligned} \tag{3}$$

2.2 Preliminary Experiment

Prior works [24, 31, 33, 39] reveal distinct learning behaviors between SFT and DPO, as illustrated in Fig. 1. Specifically, confidence scores for both $\mathbf{y}^+$ and $\mathbf{y}^-$ gradually decrease during DPO training, while $\mathbf{y}^+$ confidence remains stable under SFT. Furthermore, they find that for some responses they track (i.e., various responses similar to $\mathbf{y}^+$ or $\mathbf{y}^-$), none of them increase during the DPO phase. Ren et al. [33] refer to this phenomenon observed in DPO as the `squeezing effect`, and the decreased probability mass is squeezed onto $\mathbf{y}^*$ (the output which was most confident before the update).

To validate their analysis in practical settings, we conduct experiments on two representative code tasks. We construct the training set $\mathcal{D}_{\text{train}}$ by randomly sampling 2500 examples from the datasets described in Section 4.2, covering code correctness and efficiency preferences. Each example comprises three components: a prompt $\mathbf{x}$, a preferred response $\mathbf{y}^+$, and a dispreferred response $\mathbf{y}^-$. During training, SFT uses only $\mathbf{x}$ and $\mathbf{y}^+$, while DPO leverages all three components. We repeat the experiments on two series of models: DeepSeek-Coder 1.3B [13] and Qwen2.5-Coder 0.5B/1.5B [16].

To analyze learning dynamics in detail, we construct a probing dataset $\mathcal{D}_{\text{prob}}$ by sampling 500 examples from $\mathcal{D}_{\text{train}}$ and augmenting each with additional response variants. For code correctness task, we include responses similar to the original preferred and dispreferred responses, denoted as $\mathbf{y}^+_{\text{sim}}$ and $\mathbf{y}^-_{\text{sim}}$. For code efficiency task, we add responses with higher efficiency ($\mathbf{y}^+_h$) and lower efficiency ($\mathbf{y}^+_l$) than $\mathbf{y}^+$, reflecting real-world scenarios where the preferred response may not represent the optimal solution. We fine-tune the models for several epochs, evaluating predictions on all responses in $\mathcal{D}_{\text{prob}}$ every 100 updates. For each response type, we compute the average log-probability across all 500 examples as a measure of model confidence. This allows us to track how $\log\pi_{\theta^t}(\mathbf{y}\mid\mathbf{x})$ evolves for different response categories throughout training.

Learning dynamics of SFT and DPO. As demonstrated in the first panel of Fig. 2 and Fig. 3, SFT consistently increases the model's confidence in $\mathbf{y}^+$ throughout training, which is expected given that the optimization directly applies "pull-up" pressure to the preferred response. This increase in $\mathbf{y}^+$ probability naturally diminishes confidence in all other responses $\mathbf{y}\neq\mathbf{y}^+$, as the model's predicted probabilities across the entire $\mathcal{Y}$-space must normalize to one. In contrast, the second panel reveals that DPO exhibits markedly different behavior: confidence scores for all $\mathbf{y}$ responses decrease during training, with $\mathbf{y}^-$ and its similar variants experiencing the most significant drops, as the decreased probability mass is squeezed onto $\mathbf{y}^*$ (Argmax). Notably, the third panel demonstrates that even when DPO is applied after SFT training, the confidence in $\mathbf{y}^+$ still decreases. These empirical observations align consistently with the phenomena reported by Ren et al. [33].

3 Insights of SFT and DPO in Practical Usage

This section highlights the respective strengths of DPO and SFT in aligning with human code preferences. We first conduct a theoretical analysis of how $\mathbf{y}^+$ and $\mathbf{y}^-$ probabilities evolve when the training objectives of SFT and DPO are optimized. We then present a synthetic example to further illustrate their differences. Building upon the preliminary experiments from Section 2.2, we systematically evaluate the effectiveness of both methods across different types of human code

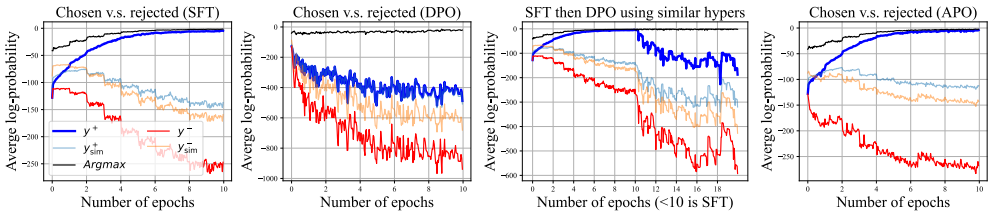

Fig. 2. Learning dynamics of SFT, DPO, and APO on code correctness preference

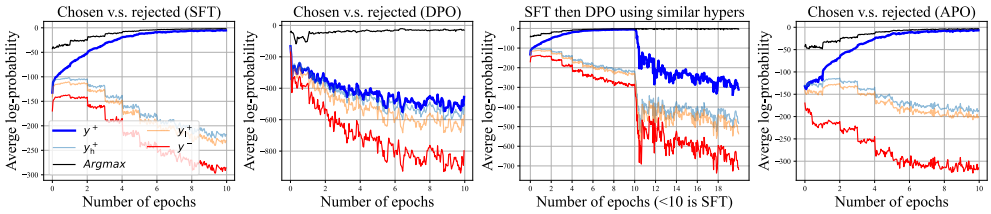

Fig. 3. Learning dynamics of SFT, DPO, and APO on code efficiency preference

preferences. Finally, based on the analysis and experimental evidence, we design a new training method to align with various human code preferences.

3.1 Theoretical Analysis

Theorem 3.1. *We analyze the objectives of SFT and DPO under ideal optimization conditions to understand their fundamental differences. Assume that both optimization processes converge to their respective global minima.*

Given a preference dataset $\mathcal{D}$, let Π_{SFT} and Π_{DPO} denote the sets of optimal policies obtained by minimizing the SFT loss in Eq. 2 and the DPO loss in Eq. 3, respectively. SFT directly maximizes the likelihood of preferred responses, while DPO optimizes relative preferences between response pairs, enhancing the model's ability to distinguish superior responses from multiple candidates. Under perfect optimization, we establish the following theoretical results: ❶ *When $\mathcal{L}_{\text{SFT}}(\pi_\theta) \to 0$, we have $\pi_\theta(\mathbf{y}^+ \mid \mathbf{x}) \to 1$ and consequently $\pi_\theta(\mathbf{y}^- \mid \mathbf{x}) \to 0$;* ❷ *When $\mathcal{L}_{\text{DPO}}(\pi_\theta) \to 0$, we have $\pi_\theta(\mathbf{y}^- \mid \mathbf{x}) \to 0$, but $\pi_\theta(\mathbf{y}^+ \mid \mathbf{x})$ does not necessarily approach 1.*

Proof. We first prove the conclusion ❶. The SFT objective minimizes the negative log-likelihood of preferred responses, thereby encouraging the model to assign maximal probability to these responses. Formally,

$$\begin{aligned} \lim_{\mathbf{x}\to\mathbf{x}_0} \mathcal{L}_{\text{SFT}}(\pi_\theta) = \lim_{\mathbf{x}\to\mathbf{x}_0} -\mathbb{E}_{(\mathbf{x},\mathbf{y}^+)\sim\mathcal{D}} \left[\log \pi_\theta(\mathbf{y}^+ \mid \mathbf{x})\right] = 0 \\ \Rightarrow \pi_\theta(\mathbf{y}^+ \mid \mathbf{x}) \to 1 \\ \Rightarrow \pi_\theta(\mathbf{y}^- \mid \mathbf{x}) \to 0. \end{aligned} \tag{4}$$

This result follows directly from the properties of the cross-entropy loss: the loss approaches zero if and only if the model assigns a probability approaching one to the target label (i.e., the preferred response $\mathbf{y}^+$).

Next, we present a counter-example in Table 1 to demonstrate the conclusion ❷. Consider a scenario with three possible responses $\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3\}$, where our preference dataset $\mathcal{D}$ contains only a single comparison pair $(\mathbf{y}^+, \mathbf{y}^-) = (\mathbf{y}_1, \mathbf{y}_2)$. Let $a = \pi_{\text{ref}}(\mathbf{y}^+ \mid \mathbf{x})$ and $b = \pi_{\text{ref}}(\mathbf{y}^- \mid \mathbf{x})$ denote the

Table 1. A counter-example with three actions

Policy	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$
π_{ref}	0.3	0.2	0.5
$\mathcal{D}_{\text{pref}}$	$\{(\mathbf{y}^+ = \mathbf{y}_1, \mathbf{y}^- = \mathbf{y}_2)\}$		
$\pi_{\text{DPO}}, \mathcal{L}_{\text{DPO}} = 0$	0.2	0.0	0.8
$\mathcal{L}_{\text{SFT}} \neq 0$	0.2	0.0	0.8
$\pi_{\text{SFT}}, \mathcal{L}_{\text{SFT}} = 0$	1.0	0.0	0.0

reference policy probabilities. From the DPO loss in Eq. 3, when the loss approaches zero, we have:

$$\lim_{\mathbf{x}\to\mathbf{x}_0} \mathcal{L}_{\text{DPO}}(\pi_\theta) = \lim_{\mathbf{x}\to\mathbf{x}_0} \mathbb{E}_{(\mathbf{x},\mathbf{y}^+,\mathbf{y}^-)\sim\mathcal{D}} \left[\log\left(1 + \left(\frac{b\cdot\pi_\theta(\mathbf{y}^- \mid \mathbf{x})}{a\cdot\pi_\theta(\mathbf{y}^+ \mid \mathbf{x})}\right)^\beta\right)\right] = 0$$

$$\Rightarrow \frac{b\cdot\pi_\theta(\mathbf{y}^- \mid \mathbf{x})}{a\cdot\pi_\theta(\mathbf{y}^+ \mid \mathbf{x})} \to 0. \tag{5}$$

Since a and b are fixed positive constants for all $(\mathbf{x}, \mathbf{y}^+, \mathbf{y}^-) \in \mathcal{D}$, the above condition directly implies $\pi_\theta(\mathbf{y}^- \mid \mathbf{x}) \to 0$. However, minimizing $\mathcal{L}_{\text{DPO}}(\pi_\theta)$ to zero does not require $\pi_\theta(\mathbf{y}^+ \mid \mathbf{x}) \to 1$. To illustrate this key difference, consider the optimal DPO policy shown in the third row of Table 1, which assigns probability 0.2 to $\mathbf{y}_1$ and 0.8 to $\mathbf{y}_3$. This policy achieves $\mathcal{L}_{\text{DPO}} = 0$ but would be impossible under SFT, which enforces $\pi_{\text{SFT}}(\mathbf{y}_1 \mid \mathbf{x}) = 1$ according to Eq. 2.

3.2 Analysis in Human Code Preferences

In this section, we analyze the objectives of SFT and DPO in practical code generation tasks. We categorize the human code preferences into two cases:

- ❶ **Preferences with objectively verifiable optimal solutions.** These correspond to tasks where a definitive correct or optimal solution can be determined through objective evaluation. Code correctness exemplifies this category, as solutions can be rigorously verified against specifications or comprehensive test suites to confirm functional correctness.
- ❷ **Preferences without objectively verifiable optimal solutions.** These involve tasks where optimality is subjective or difficult to ascertain. Code efficiency represents this category, as determining the most efficient solution is often challenging—superior alternatives may exist but remain undiscovered or require prohibitive costs to identify.

For type ❶, we demonstrate that SFT is more effective than DPO. In such cases, the model can be optimized directly using SFT alone, without requiring additional DPO training, which may provide limited additional benefits and may even harm performance. Consider the example in Table 1, where the optimization objective is to maximize code correctness (i.e., pass rate). Any solution that successfully passes all test cases represents an objectively optimal solution. As illustrated in the second and third panels of Fig. 2, DPO training reduces the probabilities assigned to both $\mathbf{y}^+$ and $\mathbf{y}^-$, squeezing the probability mass onto $\mathbf{y}^*$. The third row of Table 1 demonstrates a DPO-optimal policy that assigns a probability of 0.2 to $\mathbf{y}_1$ and 0.8 to $\mathbf{y}_3$. Under this configuration, π_{DPO} fails to achieve superior performance and actually diminishes the likelihood of generating $\mathbf{y}_1$, while π_{SFT} guarantees a probability of 1.

For type ❷, the preferred response $\mathbf{y}^+$ in the preference dataset may not represent the optimal solution (e.g., a more efficient response $\mathbf{y}$ may exist but be undiscovered by humans). Consider the example in Table 1, where the optimization objective is to maximize code efficiency. SFT drives the model toward assigning a probability of 1 to $\mathbf{y}_1$; however, this does not guarantee optimality, as a

more efficient solution may exist but remain undiscovered by humans. In contrast, DPO mitigates this limitation by enabling continued exploration, since it does not require the model to assign all probability mass to the demonstrated preferred response $\mathbf{y}_1$. In this context, SFT and DPO provide complementary advantages: SFT rapidly elevates the model's baseline capabilities to match the quality of demonstrated solutions in the preference dataset, while DPO enables the model to potentially surpass these demonstrated solutions through continued exploration. As illustrated in the second and third panels of Fig. 3, the probability mass is still squeezed onto $\mathbf{y}^*$. When $\mathbf{y}^*$ represents a more efficient solution, DPO can facilitate the model's achievement of a higher performance ceiling.

Hypothesis. *(1) In type ❶, SFT suffices to achieve optimal solutions, while DPO provides limited additional benefits and may even harm performance. (2) In type ❷, sequential application of SFT followed by DPO (S&D) achieves optimal performance, where SFT rapidly builds fundamental capabilities and DPO subsequently enables exploration of superior solutions.*

Based on the above analysis, we formulate these hypotheses and will empirically validate them in Section 5. Our goal is to investigate whether our hypotheses hold in these two types and to guide training strategy selection that enables the model to achieve superior performance.

3.3 Adaptive Preference Optimization

While SFT effectively increases the probability assigned to preferred responses ($\mathbf{y}^+$), DPO enables exploration of superior solutions and effectively reduces the probability of dispreferred responses ($\mathbf{y}^-$), though it may also inadvertently decrease the probability of preferred responses ($\mathbf{y}^+$). Since SFT and DPO demonstrate distinct advantages across different types, we propose a unified approach that eliminates the need for manual type discrimination while simultaneously boosting the probability of preferred responses ($\mathbf{y}^+$), suppressing dispreferred responses ($\mathbf{y}^-$), and encouraging exploration of potentially superior solutions. We term this dynamic integration framework **A**daptive **P**reference **O**ptimization (APO), which adaptively combines both methods through a dynamic loss:

$$\mathcal{L}_{\text{APO}}(\pi_\theta) = \alpha\, \mathcal{L}_{\text{DPO}}(\pi_\theta) \;+\; (1-\alpha)\, \mathcal{L}_{\text{SFT}}(\pi_\theta), \tag{6}$$

where $\alpha = e^{\frac{1}{T}\sum_{t=1}^{T} \log \pi_\theta(\mathbf{y}_t|\mathbf{y}_{<t})}$ denotes the geometric mean of the model's probability of generating the sequence. When α is low, the SFT term dominates, enabling rapid convergence toward $\mathbf{y}^+$. As α increases, the DPO term takes precedence, guiding the model to refine its policy and exploit more nuanced preference information. In the early stages of training, the model rarely assigns high probability to any preferred response, resulting in $\mathcal{L}_{\text{APO}} \approx \mathcal{L}_{\text{SFT}}$. As learning progresses and positive examples gain probability mass, $\mathcal{L}_{\text{APO}} \approx \mathcal{L}_{\text{DPO}}$, enabling the model to continue exploring superior solutions. This adaptive strategy effectively combines the advantages of both SFT and DPO. It leverages the direct maximum-likelihood signal to rapidly initiate learning, and as the model's outputs increasingly align with human preferences, it further reduces the probability of dispreferred responses. As demonstrated in Fig. 2 and Fig. 3, APO effectively increases the probability of generating $\mathbf{y}^+$ in both types while rapidly reducing the probability of $\mathbf{y}^-$ during the early training stages. In the code efficiency preference type, APO achieves a higher probability for $\mathbf{y}_h^+$ compared to SFT alone, demonstrating that the model continues to explore superior solutions.

4 Experimental Design

In this section, we conduct comprehensive experiments on preference datasets to investigate the effectiveness of SFT and DPO across diverse code preference types. We present our studied code preference types, constructed datasets, studied LLMs, evaluation metrics, and experimental settings.

4.1 Studied Code Preference Types

For preferences with objectively verifiable optimal solutions, we consider three types of human code preference types:

- **Code Correctness.** Ensuring that code executes correctly and produces expected outputs according to specified input-output requirements.
- **Code Security.** Beyond correctness, humans prefer generated code with minimal vulnerabilities, ensuring safe handling of sensitive data and protection against security threats.
- **Code Smell.** It refers to any symptoms in the code that may lead to deeper issues. Humans prefer well-structured code that establishes best practices and maintains high readability.

For preferences without objectively verifiable optimal solutions, we examine three additional code preference types:

- **Code Efficiency.** When multiple solutions achieve identical functionality, humans favor more efficient solutions that execute faster and consume fewer computational resources.
- **Code Complexity.** This evaluates the complexity of the code by measuring the number of linearly independent paths through the program. Humans prefer simpler, more understandable code that facilitates maintenance and testing.
- **Code Conciseness.** Humans prefer concise codes for their enhanced readability and comprehensibility. Concise code typically reduces maintenance overhead, minimizes error introduction, and promotes effective developer collaboration.

4.2 Constructed Dataset

We construct our preference dataset using the widely adopted APPS dataset [14]. The APPS dataset comprises 10,000 coding problems sourced from various open-access coding platforms, including Codeforces and Kattis. These problems span difficulty levels from introductory to collegiate competition level, designed to assess both coding proficiency and problem-solving capabilities. Each problem is presented in unrestricted natural language, closely reflecting real-world programming scenarios. The dataset includes 131,836 test cases for solution verification and 232,444 ground-truth solutions authored by human programmers. On average, each problem contains 293.2 words and is accompanied by comprehensive test cases (21.2 per problem in the test set), specifically designed to rigorously evaluate program functionality. The dataset is evenly partitioned into training and test sets, each containing 5,000 problems.

Table 2. Statistics of the constructed dataset

Dataset	# Problem	# Solution (Preference)	Optimal Solution
Test Set	500	-	-
Augmented	4,500	120,833	-
Correctness	3,868	3,868	✓
Security	3,145	3,145	✓
Smell	2,809	9,077	✓
Efficiency	3,358	3,358	✗
Complexity	3,392	3,392	✗
Conciseness	3,349	3,349	✗

Since problems in the APPS training set contain relatively few test cases, making functional correctness verification challenging, we perform three systematic steps on the test set to construct high-quality preference datasets. The statistics for each step are presented in Table 2.

Step 1: We randomly sample 10% of problems from the test set to create a new test set, resulting in 500 test problems.

Step 2: For the remaining 4,500 problems, we utilize both the original solutions from the APPS dataset and generate additional high-quality solutions using three state-of-the-art code generation models: `DeepSeek-Coder 33B` [13], `CodeLlama 34B` [34], and `Qwen2.5-Coder 32B` [16]. We generate 10 solutions per problem from each model, ensuring diverse solution coverage.

Step 3: All generated solutions undergo rigorous evaluation using the corresponding test cases from the APPS dataset. For preferences other than code correctness, solutions failing to pass all test cases are systematically filtered out, yielding a total of 120,833 functionally correct solutions. In contrast, for the code correctness, we deliberately retain these failed solutions as negative examples $\mathbf{y}^-$ during training.

To evaluate the effectiveness of SFT, DPO, and APO, we construct specialized preference training sets for each code preference type. Following prior works [53, 54], we evaluate the LLM-generated solutions using specific metrics and select preferred and rejected samples to form preference pairs:

- **Code correctness:** We employ a PageRank-inspired [19] iterative algorithm to rank code solutions. Initially, each solution receives a validation score of 1. For each coding problem, we compute scores for solutions and test cases based on their mutual performance: test cases with fewer passing solutions receive higher scores, while those with more passing solutions receive lower scores. Conversely, solutions passing more test cases receive higher scores than those passing fewer test cases. These scores are iteratively updated over T = 5 iterations based on validation performance. Let $P \in \{0,1\}^{n\times m}$ denote the binary pass matrix where $P_{ij} = 1$ indicates solution i passes test case j, and $F = \mathbf{1} - P$ represents the corresponding fail matrix. Let $\mathbf{s} \in \mathbb{R}^n$ and $\mathbf{t} \in \mathbb{R}^m$ denote the score vectors for solutions and test cases, respectively. The iterative update process follows:

$$\begin{aligned}\mathbf{s}^{(k+1)} &= \alpha \times P \times \mathbf{t}^{(k)} + (1-\alpha) \times \mathbf{s}^{(k)} \\ \mathbf{t}^{(k+1)} &= \alpha \times F^\top \times \mathbf{s}^{(k)} + (1-\alpha) \times \mathbf{t}^{(k)},\end{aligned} \tag{7}$$

  where $\alpha \in (0,1)$ is a damping factor controlling the balance between newly propagated scores and previous scores. After each iteration, $\mathbf{s}^{(k+1)}$ and $\mathbf{t}^{(k+1)}$ are normalized to ensure $\sum_i s_i = n$ and $\sum_j t_j = m$.
  For each problem, we select the highest and lowest scoring solutions from all successfully compiled codes, yielding 3,868 preference pairs.
- **Code security:** Since APPS primarily focuses on algorithmic problems and rarely contains inherent security vulnerabilities, we utilize injection methods from ProsSec [47] to systematically introduce vulnerabilities into solutions. We construct preference pairs by contrasting vulnerable and non-vulnerable solution variants, resulting in 3,145 preference pairs. Evaluation is conducted using the CyberSec [42] benchmark.
- **Code smell:** Following previous works [17, 36, 41], we employ the Pylint static analysis tool for smell detection. We apply Pylint to each solution, identifying code smells requiring refactoring. For each detected smell type, we randomly pair the corresponding solution with a smell-free counterpart, generating 9,077 preference pairs.
- **Code efficiency:** Following the methodology of [28], we utilize the Cirron [37] library to measure CPU instruction counts. Unlike execution time measurements, which can be influenced by external factors, CPU instruction counts provide stable, deterministic performance metrics. We select the most and least efficient solutions for each problem, creating 3,358 preference pairs.

- **Code complexity:** We employ the Radon [2] tool to calculate cyclomatic complexity, selecting solution pairs with maximum and minimum complexity scores. This process yields 3,392 preference pairs.
- **Code conciseness:** We calculate token lengths for each solution and pair the longest and shortest implementations per problem, resulting in 3,349 preference pairs.

4.3 Studied Large Language Models

We evaluate four series of open-source LLMs: `Llama 3.2 1B` [22], `DeepSeek-Coder 1.3B/6.7B` [13], `Magicoder 6.7B` [45], and `Qwen2.5-Coder 1.5B/7B` [16]. Parameter count constraints are imposed by our available computational resources (4 × NVIDIA A800 GPUs).

4.4 Evaluation Metrics

Inspired by *Pass@k* [5] and *Efficient@k* [28], we further extend *{Preference}@k* to comprehensively evaluate various code preference types. The definition of *{Preference}@k* is formalized in Eq. 8.

$$\{\text{Preference}\}@k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c_p}{k}}{\binom{n}{k}} \right], \tag{8}$$

where c_p represents the number of solutions that pass all test cases while satisfying the specific preference criterion: being free of code smells, more efficient, less complex, or more concise than the APPS-provided baseline solution. With values ranging from 0 to *Pass@k*, *{Preference}@k* integrates functional correctness with task-specific evaluation for comprehensive code quality assessment. For specific criteria, we define corresponding metrics: *Clean@k* for code smell, *Simple@k* for code complexity, and *Concise@k* for code conciseness.

Beyond *{Preference}@k*, we evaluate *{Preference} Rate*, defined as the proportion of solutions that pass all test cases while satisfying the specific preference criterion. We define corresponding metrics: *Clean Rate*, *Efficiency Rate*, *Simplicity Rate*, and *Conciseness Rate* for their respective criteria. For code security, we report *Security Rate* as measured on the CyberSec [42] benchmark.

4.5 Implementation

We implement the generation pipeline in Python using PyTorch [27] implementations of `Llama 3.2`, `DeepSeek-Coder`, `Magicoder`, and `Qwen2.5-Coder`. Model weights are loaded and outputs generated via Hugging Face [1] libraries. For SFT, DPO, and APO training, we select models with at most 7B parameters due to computational constraints. To prevent overfitting, we limit training to 5 epochs. For evaluation, both original and fine-tuned models generate 5 solutions per test problem. Experiments are conducted on a 32-core workstation equipped with Intel(R) Xeon(R) Platinum 8358P CPU @ 2.60GHz, 2TB RAM, and 4×NVIDIA A800 80GB GPUs, running Ubuntu 20.04.6 LTS.

5 Results

To investigate the effectiveness of SFT, DPO, and APO in aligning human code preferences, we conduct comprehensive experiments addressing the following three research questions:

- *RQ-1 How well do SFT and DPO perform on types with objectively verifiable optimal solutions?*
- *RQ-2 How well do SFT and DPO perform on types without objectively verifiable optimal solutions?*
- *RQ-3 Can APO achieve comparable performance to SFT and S&D on code preference types?*

5.1 RQ-1: Comparable Study of SFT and DPO on Type ❶

Objective. With the continuous advancement of deep learning technologies, LLMs have become essential tools for code generation. SFT and DPO are two prominent training strategies that have

demonstrated promising results in various code generation tasks [49, 53, 55]. In Section 3, we hypothesize that for type ❶, SFT is sufficient to achieve optimal performance, while DPO provides limited additional benefits and may even harm performance. This research question aims to provide a comprehensive empirical evaluation of SFT and DPO across different code preference types (i.e., code correctness, code security, and code smell). Through this empirical evaluation, we aim to validate our hypothesis (i.e., SFT is sufficient to achieve optimal performance for type ❶) and determine which strategy offers superior practical advantages.

Experimental Design. We establish baselines using various original LLMs, including `Llama 3.2 1B` (referred to as Llama), `DeepSeek-Coder 1.3B/6.7B` (referred to as DeepSeek), `Magicoder 6.7B` (referred to as Magicoder), and `Qwen2.5-Coder 1.5B/7B` (referred to as Qwen). These models are subsequently fine-tuned using both SFT and DPO strategies, leveraging the datasets constructed in Section 4.2. Additionally, we employ a sequential training strategy, performing SFT followed by DPO (referred to as S&D). Each example in $\mathcal{D}$ consists of three components: the prompt (or question) $\mathbf{x}$, the preferred response $\mathbf{y}^+$, and the less preferred response $\mathbf{y}^-$. SFT is fine-tuned using $\mathbf{x}$ and $\mathbf{y}^+$, while DPO utilizes all three components. For code correctness, we report the overall *Pass@5* metric and *Pass@5* metrics segmented by difficulty levels: competition, interview, and introductory. For code smell, we calculate *Clean@5* and *Clean Rate*. For code security, we report the *Security Rate* as measured on the CyberSec [42] benchmark.

Results. We present and analyze the results for code correctness, security, and smell preferences, respectively.

Code Correctness Preference. The results for code correctness are presented in Table 3, with the best performances highlighted in bold. Based on these results, we observe that the SFT training strategy outperforms the DPO training strategy, achieving superior results across most metrics. Specifically, SFT achieves Pass@5 scores ranging from 6.2% to 27.0%, representing improvements of 1.0% to 6.2% over the original models. In contrast, DPO yields a lower Pass@5 range of 2.6% to 25.4%. Notably, for DeepSeek-Coder 1.3B, DPO not only fails to enhance the original model's capabilities but actually reduces its functional accuracy. Furthermore, applying DPO after an initial SFT phase (S&D) does not consistently yield significant advantages over SFT alone; in some cases, it can even diminish SFT's effectiveness. For instance, on the Qwen2.5-Coder 7B model, the S&D strategy achieves a Pass@5 of only 25.8%, which is inferior to the 27.0% achieved by SFT alone.

Table 3. RQ-1: The effectiveness of SFT and DPO in improving code correctness

LLM	Method	Pass@5	Comp.	Inter.	Intro.	LLM	Method	Pass@5	Comp.	Inter.	Intro.
DeepSeek 1.3B	Ori.	8.6%	1.0%	6.6%	**22.0%**	DeepSeek 6.7B	Ori.	17.8%	**5.1%**	15.2%	38.0%
	SFT	**9.6%**	0.0%	**8.6%**	**22.0%**		SFT	**22.0%**	4.1%	18.5%	**50.0%**
	DPO	8.0%	**3.1%**	6.3%	18.0%		DPO	18.6%	**5.1%**	15.6%	41.0%
	S&D	9.0%	2.0%	7.9%	19.0%		S&D	**22.0%**	4.1%	**20.5%**	44.0%
Qwen 1.5B	Ori.	9.4%	1.0%	6.6%	26.0%	Qwen 7B	Ori.	20.8%	4.1%	19.9%	40.0%
	SFT	**14.8%**	**3.1%**	**13.2%**	**31.0%**		SFT	**27.0%**	**13.3%**	**25.5%**	**45.0%**
	DPO	11.0%	0.0%	9.6%	26.0%		DPO	25.4%	12.2%	24.2%	42.0%
	S&D	14.6%	**3.1%**	12.9%	**31.0%**		S&D	25.8%	10.2%	24.5%	**45.0%**
Llama 1B	Ori.	1.4%	0.0%	1.3%	3.0%	Magicoder 6.7B	Ori.	16.0%	5.1%	13.9%	33.0%
	SFT	**6.2%**	**1.0%**	**5.3%**	**14.0%**		SFT	**22.0%**	5.1%	**19.5%**	**46.0%**
	DPO	2.6%	0.0%	1.7%	8.0%		DPO	18.2%	4.1%	15.6%	40.0%
	S&D	3.8%	0.0%	3.0%	10.0%		S&D	20.6%	**7.1%**	17.5%	43.0%

Code Security Preference. The effectiveness of SFT and DPO in enhancing code security is detailed in Table 4. Based on these results, we can make several key observations: (1) **SFT significantly enhances code security.** Across various LLMs, SFT training typically elevates baseline security rates

Table 4. RQ-1: The effectiveness of SFT and DPO in improving security rate

LLM	Ori.	SFT	DPO	S&D	LLM	Ori.	SFT	DPO	S&D
Llama 1B	77.4%	90.3%	80.9%	**91.2%**	Magicoder 6.7B	71.2%	84.2%	72.4%	**85.1%**
DeepSeek 1.3B	71.7%	85.7%	73.3%	**86.4%**	DeepSeek 6.7B	72.1%	**84.9%**	72.7%	84.7%
Qwen 1.5B	75.3%	**88.9%**	76.6%	88.3%	Qwen 7B	70.2%	**86.8%**	71.9%	86.4%

from approximately 70.2%-75.3% to a substantially improved range of 84.2%-90.3%. In comparison, while DPO alone offers some improvements over the original models, SFT consistently provides more substantial security enhancements. (2) **Subsequent DPO training after SFT offers limited additional benefits and can occasionally be detrimental.** LLMs trained with SFT achieve security rates ranging from 84.2% to 90.3%. After subsequent DPO training, these rates adjust to a range of 84.7% to 91.2%. Notably, in some cases, the S&D training leads to slight reductions in overall security rates. For example, Qwen2.5-Coder 1.5B trained with SFT achieves a security rate of 88.9%, which decreases to 88.3% after subsequent DPO training.

We further compare SFT and S&D across different Common Weakness Enumerations (CWEs). We focus on the five most frequent CWEs in the CyberSec [42] benchmark: CWE-78, CWE-94, CWE-328, CWE-502, and CWE-798. As shown in Table 5, SFT and S&D each demonstrate advantages on different CWEs, but the overall differences are not significant.

Table 5. RQ-1: The performance comparison among six LLMs on Top-5 CWEs (Security / Total)

CWE	Llama 1B		DeepSeek 1.3B		Qwen 1.5B	
	SFT	S&D	SFT	S&D	SFT	S&D
CWE-78	373/415	376/415	367/415	373/415	388/415	386/415
CWE-94	300/325	295/325	244/325	254/325	279/325	271/325
CWE-328	256/275	269/275	262/275	270/275	265/275	264/275
CWE-502	190/215	194/215	182/215	187/215	181/215	181/215
CWE-798	162/185	166/185	173/185	162/185	168/185	166/185
CWE	**Magicoder 6.7B**		**DeepSeek 6.7B**		**Qwen 7B**	
	SFT	**S&D**	**SFT**	**S&D**	**SFT**	**S&D**
CWE-78	383/415	371/415	371/415	373/415	380/415	377/415
CWE-94	250/325	261/325	266/325	259/325	280/325	272/325
CWE-328	235/275	246/275	233/275	242/275	254/275	264/275
CWE-502	182/215	180/215	180/215	181/215	174/215	175/215
CWE-798	156/185	161/185	168/185	159/185	153/185	153/185

Code Smell Preference. The effectiveness of SFT and DPO in reducing code smell is reported in Table 6. Based on these results, we observe the following key findings: (1) **SFT consistently outperforms DPO in reducing code smell.** For example, with the Llama 3.2 1B model, SFT training enhances the Clean@5 score from 1.2% to 4.6%, whereas DPO training only increases it to 2.8%. Regarding clean rate, SFT training boosts it from 80.0% to 95.5%, while DPO training only increases it to 81.8%. (2) **Applying DPO training after SFT often proves counterproductive.** LLMs trained with SFT achieve clean rates ranging from 91.2% to 96.6%; however, the S&D training reduces this range to 88.4% to 95.4%. (3) **Overall, SFT demonstrates substantial improvements across all models.** SFT improves Clean@5 scores from an initial range of 1.2%-18.0% to a resulting range of 4.6%-23.6%, and enhances clean rates from 75.6%-85.7% to 90.1%-96.6%.

Answer to RQ-1: *For type ❶, SFT is sufficient for achieving strong performance, while S&D may not provide additional benefits and may even lead to diminished results.*

Table 6. RQ-1: The effectiveness of SFT and DPO in reducing code smell

LLM	Clean@5				Clean Rate			
	Ori.	SFT	DPO	S&D	Ori.	SFT	DPO	S&D
Llama 1B	1.2%	**4.6%**	2.8%	3.6%	80.0%	**95.5%**	81.8%	95.4%
DeepSeek 1.3B	7.4%	**8.2%**	6.0%	7.4%	75.6%	**90.1%**	74.6%	89.3%
Qwen 1.5B	8.0%	**11.8%**	8.8%	10.8%	81.2%	**96.6%**	82.8%	93.4%
Magicoder 6.7B	15.0%	**20.4%**	13.4%	19.8%	85.7%	**92.4%**	80.7%	88.4%
DeepSeek 6.7B	15.0%	**18.2%**	14.8%	17.4%	83.8%	91.2%	83.6%	**92.0%**
Qwen 7B	18.0%	**23.6%**	23.0%	22.6%	82.1%	**94.8%**	84.9%	94.7%

5.2 RQ-2: Comparable Study of SFT and DPO on Type ❷

Objective. In Section 3, we hypothesize that for type ❷, sequential application of SFT followed by DPO (S&D) achieves optimal performance, where SFT rapidly builds fundamental capabilities and DPO subsequently enables exploration of superior solutions. This research question aims to empirically evaluate SFT and DPO in improving code efficiency, code complexity, and code conciseness. Through this evaluation, we seek to validate our hypothesis (i.e., S&D achieves optimal performance in type ❷) and identify which strategy offers greater practical advantages.

Experimental Design. The models and training procedures used in RQ-2 are consistent with those in RQ-1 (refer to Section 5.1 for detailed specifications). The evaluation metrics for code efficiency, code complexity, and code conciseness are *Efficient@5*, *Simple@5*, and *Concise@5*, respectively, as defined in Section 4.5. Additionally, we calculate the *Efficiency Rate*, *Simplicity Rate*, and *Conciseness Rate*, which represent the proportion of code that meets the corresponding preference standards among the solutions that pass all test cases.

Results. We present and analyze the results for code efficiency, complexity, and conciseness preferences, respectively.

Table 7. RQ-2: The effectiveness of SFT and DPO in improving code efficiency

LLM	Efficient@5				Efficiency Rate			
	Ori.	SFT	DPO	S&D	Ori.	SFT	DPO	S&D
Llama 1B	0.8%	**3.0%**	0.8%	2.8%	50.0%	85.7%	57.1%	**93.8%**
DeepSeek 1.3B	6.4%	7.0%	6.6%	**7.4%**	56.4%	77.1%	**85.9%**	80.7%
Qwen 1.5B	7.8%	9.8%	9.6%	**10.0%**	74.1%	85.9%	83.3%	**90.4%**
Magicoder 6.7B	14.0%	**21.0%**	14.2%	18.4%	77.1%	83.7%	78.2%	**85.3%**
DeepSeek 6.7B	15.4%	17.6%	16.6%	**17.8%**	80.9%	**84.7%**	81.3%	81.0%
Qwen 7B	18.4%	18.4%	17.8%	**18.8%**	80.8%	84.5%	78.0%	**85.9%**

Code Efficiency Preference. The impact of SFT, DPO, and S&D training on improving code efficiency is detailed in Table 7. We observe that models first establish foundational capabilities in generating efficient code after SFT training. For instance, SFT substantially improves both Efficient@5 and efficiency rate metrics across all models compared to their original baselines (e.g., for Llama 1B, Efficient@5 improves from 0.8% to 3.0%, and efficiency rate increases from 50.0% to 85.7%). Building upon this SFT-established foundation, the subsequent application of DPO training, as part of the S&D training, appears to further encourage the models to explore and generate more efficient code. Consequently, the S&D training frequently achieves superior performance, demonstrating advantages in both Efficient@5 and efficiency rate metrics across several LLMs.

Code Complexity Preference. Table 8 summarizes the effectiveness of SFT and DPO in reducing code complexity, as measured by cyclomatic complexity. Both SFT and S&D training substantially

improve Simple@5 scores compared to baseline models. Although SFT achieves superior performance on Simple@5 metrics, it does not demonstrate significant improvements in simplicity rate. We attribute this phenomenon to the influence of code correctness requirements, as Simple@5 requires solutions to be both simple and pass all test cases, while DPO may reduce the pass rate of generated code. Nevertheless, S&D achieves higher simplicity rates, indicating that among solutions passing all test cases, those generated by S&D generally exhibit lower complexity.

Table 8. RQ-2: The effectiveness of SFT and DPO in reducing cyclomatic complexity

LLM	Simple@5				Simplicity Rate			
	Ori.	SFT	DPO	S&D	Ori.	SFT	DPO	S&D
Llama 1B	0.2%	**5.2%**	1.4%	4.2%	10.0%	**100.0%**	35.0%	**100.0%**
DeepSeek 1.3B	3.2%	**9.4%**	5.0%	8.0%	26.9%	96.3%	48.5%	**97.1%**
Qwen 1.5B	3.0%	11.6%	6.4%	**12.4%**	28.2%	**98.9%**	58.1%	98.3%
Magicoder 6.7B	8.2%	**21.8%**	12.4%	19.8%	45.7%	96.9%	68.9%	**97.5%**
DeepSeek 6.7B	7.4%	**20.0%**	12.2%	19.8%	33.8%	98.7%	59.9%	**98.0%**
Qwen 7B	6.6%	**22.8%**	18.6%	20.4%	30.0%	96.2%	63.2%	**97.6%**

Code Conciseness Preference. Table 9 presents the results on code conciseness, evaluating metrics such as Concise@5 and conciseness rate after applying SFT and DPO training. The findings demonstrate that while SFT significantly enhances the conciseness of code generated by baseline LLMs, the S&D training yields superior outcomes. Notably, S&D generally surpasses SFT in improving Concise@5 scores across several models. Furthermore, S&D particularly excels in maximizing the conciseness rate, achieving top performance for all tested LLMs, with rates often approaching near-optimal levels (ranging from 93.0% to 100.0%). This underscores the effectiveness of the combined SFT and DPO training in guiding LLMs to produce highly concise code solutions.

Table 9. RQ-2: The effectiveness of SFT and DPO in improving code conciseness

LLM	Concise@5				Conciseness Rate			
	Ori.	SFT	DPO	S&D	Ori.	SFT	DPO	S&D
Llama 1B	0.4%	**4.0%**	1.4%	3.8%	20.0%	96.7%	42.1%	**100.0%**
DeepSeek 1.3B	5.4%	**6.6%**	4.2%	6.4%	51.3%	86.8%	51.8%	**97.2%**
Qwen 1.5B	5.0%	8.0%	5.8%	**8.6%**	36.5%	92.1%	43.3%	**93.0%**
Magicoder 6.7B	12.0%	18.2%	13.0%	**18.6%**	65.7%	93.4%	61.1%	**95.0%**
DeepSeek 6.7B	13.6%	15.8%	14.0%	**16.4%**	66.7%	91.5%	64.6%	**93.1%**
Qwen 7B	11.4%	17.0%	16.2%	**18.2%**	43.3%	94.9%	49.7%	**97.7%**

To further demonstrate the effectiveness of SFT and DPO training strategies, we conduct an additional comparative analysis. For each LLM, we first identify the common set of problems for which its original version, as well as its variants trained with SFT, DPO, and S&D respectively, all generate solutions that successfully pass test cases. For each problem within this common set, we randomly select one solution from the pool generated by each of these four model variants (i.e., Original, SFT, DPO, and S&D). After evaluating these selected solutions based on predefined performance metrics, we calculate the total number of times each distinct approach yields the best-performing solution. As shown in Fig. 4, we observe that SFT establishes strong foundational capabilities. Subsequent DPO training (S&D) then effectively promotes further exploration and refinement of solutions. Consequently, the S&D training consistently achieves the highest number of best-performance instances, underscoring its overall superiority in these comparative evaluations.

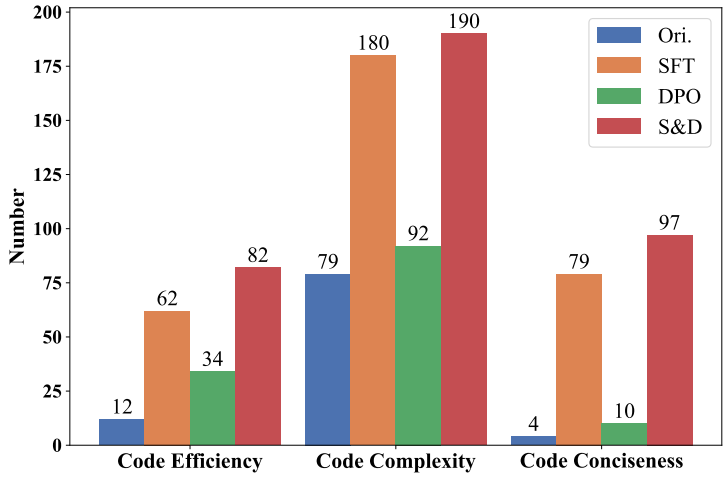

Fig. 4. RQ-2: Number of best-performance instances across code preference types

Answer to RQ-2: *For type ❷, sequential application of SFT followed by DPO (S&D) achieves optimal performance, where SFT rapidly builds fundamental capabilities and DPO subsequently enables exploration of superior solutions.*

5.3 RQ-3: Comparing APO with SFT and DPO

Objective. While SFT effectively increases the probability assigned to preferred responses (y^+), DPO enables exploration of superior solutions and effectively reduces the probability of dispreferred responses (y^-), though it may also inadvertently decrease the probability of preferred responses (y^+). Given that SFT and DPO demonstrate distinct advantages across different preference types, we propose a unified approach that eliminates the need for manual type discrimination while simultaneously boosting the probability of preferred responses (y^+), suppressing dispreferred responses (y^-), and encouraging exploration of potentially superior solutions during training. We term this dynamic integration framework **A**daptive **P**reference **O**ptimization (APO), which adaptively leverages the complementary strengths of SFT and DPO. In this research question, we aim to determine whether our APO framework offers superior performance in optimizing response preferences and fostering solution exploration when compared to standalone SFT, DPO, and their sequential S&D application.

Experimental Design. The models and metrics used in RQ-3 are consistent with those in RQ-1 and RQ-2. Considering the respective strengths of SFT and S&D in different types, we compare APO with SFT in type ❶ and with S&D in type ❷. We also evaluate the efficiency of APO, SFT, DPO, and S&D in terms of training time and GPU memory cost. For these analyses, we select three representative models: `Qwen2.5-Coder 7B`, `Magicoder 6.7B`, and `DeepSeek-Coder 6.7B`. Training time cost is measured as the average time required to complete all training epochs for each preference type, and GPU memory cost is reported as the average usage during training.

Results. We present and analyze the results from both effectiveness and efficiency perspectives, respectively.

Effectiveness Comparison. The effectiveness of APO compared against SFT and S&D is detailed in Tables 10 and Table 11. In type ❶, as shown in Table 10, APO achieves performance outcomes that are broadly comparable to SFT. While SFT tends to yield higher scores in Pass@5 and Clean@5

metrics across several models, APO demonstrates competitive or superior results in security rate and clean rate metrics.

Table 10. RQ-3: The effectiveness of APO and SFT in type ❶

LLM	Pass@5		Security Rate		Clean@5		Clean Rate	
	SFT	APO	SFT	APO	SFT	APO	SFT	APO
Llama 1B	**6.2%**	3.8%	90.3%	**90.9%**	**4.6%**	4.4%	95.5%	**100.0%**
DeepSeek 1.3B	**9.6%**	**9.6%**	**85.7%**	85.6%	**8.2%**	6.0%	90.1%	**95.2%**
Qwen 1.5B	**14.8%**	11.8%	88.9%	**89.3%**	**11.8%**	9.4%	**96.6%**	94.7%
Magicoder 6.7B	**22.0%**	**22.0%**	84.2%	**84.9%**	**20.4%**	17.0%	92.4%	**97.5%**
DeepSeek 6.7B	**22.0%**	20.8%	84.9%	**85.9%**	**18.2%**	17.4%	91.2%	**94.8%**
Qwen 7B	**27.0%**	24.8%	**86.8%**	86.5%	**23.6%**	20.0%	94.8%	**96.4%**

When evaluated against S&D in type ❷ (Table 11), APO generally exhibits more pronounced advantages across multiple evaluation metrics. APO frequently achieves higher scores in Efficient@5 (e.g., 21.9% for Magicoder 6.7B vs. S&D's 18.4%), Simple@5 (e.g., 21.8% for Qwen 7B vs. S&D's 20.4%), and Concise@5 (e.g., 19.0% for Qwen 7B vs. S&D's 18.2%). These experimental results suggest that APO training effectively promotes model exploration towards qualitatively superior solutions across preference dimensions.

Overall, APO offers a significant advantage by streamlining the training pipeline for LLMs. Its unified approach can deliver comparable or superior capabilities without requiring practitioners to distinguish between specific use-case types when choosing between SFT or S&D, thereby simplifying both training and deployment processes.

Table 11. RQ-3: The effectiveness of APO and S&D in type ❷

LLM	Efficient@5		Efficiency Rate		Simple@5		Simplicity Rate		Concise@5		Conciseness Rate	
	S&D	APO	S&D	APO	S&D	APO	S&D	APO	S&D	APO	S&D	APO
Llama 1B	2.8%	**3.1%**	93.8%	**93.9%**	4.2%	**5.2%**	**100.0%**	**100.0%**	**3.8%**	3.2%	**100.0%**	95.2%
DeepSeek 1.3B	**7.4%**	6.9%	**80.7%**	76.0%	8.0%	**9.0%**	**97.1%**	96.8%	6.4%	**7.2%**	**97.2%**	96.1%
Qwen 1.5B	10.0%	**11.2%**	90.4%	**91.7%**	**12.4%**	12.2%	98.3%	**99.3%**	8.6%	**9.4%**	93.0%	**98.0%**
Magicoder 6.7B	18.4%	**21.9%**	**85.3%**	84.3%	19.8%	**20.2%**	97.5%	**98.2%**	18.6%	**18.8%**	95.0%	**96.9%**
DeepSeek 6.7B	17.8%	**18.0%**	81.0%	**86.3%**	**19.8%**	18.8%	98.0%	**99.6%**	16.4%	**17.8%**	93.1%	**97.0%**
Qwen 7B	18.8%	**19.2%**	85.9%	**90.8%**	20.4%	**21.8%**	97.6%	**99.4%**	18.2%	**19.0%**	**97.7%**	97.1%

Efficiency Comparison. Table 12 presents detailed analyses of training time and GPU memory cost for different training strategies, including SFT, DPO, S&D, and APO. We observe that while APO generally requires more training time than SFT across the evaluated tasks, its training duration is notably similar to that of standalone DPO. More importantly, APO exhibits a clear time advantage over the S&D pipeline, completing training significantly faster across all tasks (e.g., 219 minutes for APO vs. 344 minutes for S&D on code correctness). Regarding GPU memory cost, APO's average usage (101 GB) is comparable to DPO (101 GB) and S&D (102 GB). These efficiency characteristics position APO as a practical option for practical LLM training.

Answer to RQ-3: *APO provides a unified training framework that achieves comparable or superior performance to SFT and S&D without requiring type-specific strategy selection, thereby simplifying the training pipeline. Additionally, APO demonstrates competitive efficiency in terms of training time and GPU memory cost.*

Table 12. RQ-3: The training time cost and average GPU memory cost on different tasks

Preference: Time Cost (Minutes)	**SFT**	**DPO**	**S&D**	**APO**
Code Correctness	122	219	344	219
Code Security	122	209	331	209
Code Smell	116	555	895	555
Code Efficiency	134	256	338	260
Code Complexity	134	224	364	224
Code Conciseness	166	236	398	231
Average GPU Memory Cost (GB)	76	101	102	101

6 Discussion

6.1 Human Study on Evaluation Metrics

To validate the effectiveness of proxy metrics in capturing true human preferences, we conduct a comprehensive human study. The study follows a five-step process: task selection, participant recruitment, sampling strategy, annotation process and agreement, and results analysis.

6.1.1 Task Selection. While code correctness, security, smell, and efficiency can be assessed through relatively direct and objective metrics and have been widely adopted in numerous studies [5, 8, 17, 28, 36, 42, 47, 53], subjective preferences such as complexity and conciseness are more closely tied to developer perception and cognitive load. Consequently, we conduct a human study focusing on these two preferences to validate the effectiveness of proxy metrics.

6.1.2 Participant Recruitment. To ensure the authority of our evaluation and its alignment with industrial standards, we recruit 10 professional software engineers to participate in our study. All participants possess at least 5 years of experience in software development, with expertise spanning multiple programming paradigms (e.g., object-oriented and functional programming), software design patterns, and the management of architectural complexity. This rigorous selection process ensures that the annotators have the necessary domain knowledge to accurately assess code quality, particularly with respect to complexity and conciseness.

6.1.3 Sampling Strategy. During dataset construction and result verification, LLMs generate hundreds of thousands of samples, rendering a large-scale human study extremely challenging. This necessitates a sampling strategy that balances evaluation feasibility with statistical reliability. Consequently, following prior works [9, 23, 38], we employ Cochran's formula [6] to determine the sample size. With a confidence level of 95% and a margin of error of 5%, this approach guarantees a statistically representative selection of LLM-generated solutions. We sample from all test-passing solutions generated by the LLMs trained using SFT, DPO, S&D, and APO. For code complexity, the LLMs generate 4,548 test-passing solutions, from which we sample 355 instances. For code conciseness, the LLMs generate 3,485 test-passing solutions, yielding a sample size of 347. These sample sizes strike a balance between statistical rigor and the high cost of expert annotation. After determining the sample sizes, we apply random sampling across both tasks to ensure representativeness. Each sampled solution is paired with an APPS-provided baseline solution to form an evaluation task, yielding a total of 702 tasks.

6.1.4 Annotation Process and Agreement. Two independent annotators are paired to collaboratively complete the annotation tasks, where each task consists of an APPS-provided baseline solution and an LLM-generated solution. Each annotator dedicates approximately 20 days to the annotation process, evaluating 5 to 10 tasks per day. For each task, annotators are asked to make a binary

judgment: whether the LLM-generated solution is *better* than the baseline (labeled as 1) or *comparable to or worse* than the baseline (labeled as 0). Specifically, for complexity, annotators assess whether the LLM-generated solution exhibits lower cognitive complexity; for conciseness, they evaluate whether it is more concise. To assess the reliability of the human evaluation, following prior works [10, 35, 43], we compute the inter-annotator agreement using Cohen's Kappa coefficient [7]. The resulting value of 0.88 indicates substantial agreement between annotators, confirming the consistency and trustworthiness of the human judgments. In cases of disagreement, the two annotators discuss the task collaboratively and reach a consensus decision through deliberation.

Table 13. Agreement rate and positive rate between human judgments and proxy metrics

Preference	Sample (#)	Agreement Rate (%)	Positive Rate (%)	
			Human	Proxy
Code Complexity	355	94.6%	90.1%	91.0%
Code Conciseness	347	96.0%	86.2%	85.6%

6.1.5 Results Analysis. With reliable human annotations established, we evaluate the effectiveness of proxy metrics by comparing them against the human judgments. As shown in Table 13, we report the agreement rate and positive rate. The agreement rate denotes the percentage of tasks where human and proxy predictions match, while the positive rate denotes the proportion of tasks where the LLM-generated solution is judged as better than the baseline. The high agreement rates (94.6% for complexity and 96.0% for conciseness) demonstrate strong alignment between proxy metrics and human judgments. Furthermore, the positive rates obtained from the proxy metrics closely approximate those derived from human judgments, with differences within 1% for both preferences. These results demonstrate that proxy metrics can reliably reflect human preferences, thereby validating their use as scalable alternatives to costly human evaluation.

6.2 Descriptive Statistics of Preferences

To evaluate dataset quality and APO's effectiveness, we present descriptive statistics for preferences. For code correctness, security, and smell, preference pairs are inherently categorical: $\mathbf{y}^+$ corresponds to label 1 (i.e., correct, secure, and smell-free) and $\mathbf{y}^-$ corresponds to label 0. Due to this binary nature, computing deltas between preference pairs is not applicable. In contrast, code efficiency, complexity, and conciseness are measured using continuous metrics, enabling quantitative analysis of preference gaps. Table 14 summarizes the descriptive statistics, reporting mean values across three settings: training set preference pairs ($\mathbf{y}^+$ and $\mathbf{y}^-$), test set baseline solutions provided by APPS, and test-passing solutions generated by Qwen 7B trained with APO.

Table 14. Descriptive statistics of coding preferences (mean values)

Preference	Training Set		Test Set	Qwen 7B (APO)
	$\mathbf{y}^+$	$\mathbf{y}^-$		
Code Efficiency (CPU Instruction)	873,971,687	8,072,136,128	2,571,219,295	553,454,786
Code Complexity (Cyclomatic)	0.55	14.31	2.15	0.63
Code Conciseness (Token Length)	113.93	755.94	172.50	126.93

Based on the results in Table 14, we observe two key findings. First, the substantial gaps between $\mathbf{y}^+$ and $\mathbf{y}^-$ in the training set demonstrate that our preference pairs capture distinctions in code quality. For example, the difference reaches approximately 9.3-fold in code efficiency, 26.0-fold in

code complexity, and 6.6-fold in code conciseness. Second, the APO-trained model generates solutions that consistently surpass the test set baseline across all three preferences. For code efficiency, the APO-trained model achieves a mean CPU instruction count of 553,454,786, representing a 78.5% reduction compared to the test set baseline (i.e., 2,571,219,295). Similarly, the model produces code with lower complexity (i.e., 2.15 to 0.63, a 70.7% reduction in cyclomatic complexity) and improved conciseness (i.e., 172.50 to 126.93, a 26.4% reduction in token length). These results validate that APO effectively optimizes LLMs to generate code that better aligns with human preferences.

6.3 Threats to Validity

Internal Validity. Potential threats to internal validity primarily stem from the construction of our preference dataset and the design of evaluation metrics. Although we carefully filter and validate code solutions using rigorous test cases and established tools (e.g., `Pylint` and `Cirron`), there remains a risk of undetected errors or systematic biases in labeling code preferences. Additionally, our reliance on automated tools for code smell detection and security vulnerability injection may introduce measurement inaccuracies that could affect the validity of preference annotations. To mitigate these risks, we employ multiple complementary models and evaluation metrics to triangulate results and reduce potential systematic biases. Another potential concern lies in our choice of using extreme values (i.e., maximum and minimum) for constructing preference pairs. For efficiency, complexity, and conciseness, we also construct preference pairs using the 10th and 90th percentiles. Under S&D and APO training, we observe average reductions of 3.1% and 2.5%, respectively. A possible explanation is that the gap between $\mathbf{y}^+$ and $\mathbf{y}^-$ becomes smaller, which makes exploration more difficult for the model. This is also consistent with the common belief in the community that DPO tends to work better when there is a larger preference gap between the chosen and rejected responses [18, 26]. Note that for correctness, security, and smell, preferences are inherently binary and do not have meaningful intermediate values.

External Validity. The generalizability of our findings may be constrained by several factors, including our choice of datasets, model architectures, and code preference types. Our experiments are conducted primarily on the APPS dataset with a specific set of LLMs, which may not comprehensively represent the full spectrum of real-world programming tasks and model architectures encountered in practice. To mitigate this risk, we include diverse model architectures across different parameter scales and evaluate multiple complementary preference criteria. However, future studies incorporating additional datasets, programming languages, and preference dimensions would be valuable to confirm the broader applicability and robustness of our paper.

Construct Validity. A potential threat to construct validity arises from the risk that the proxy metrics used to measure code preference scenarios may not accurately represent the underlying phenomena of interest. For example, cyclomatic complexity serves as a proxy for code complexity, but whether it fully captures the cognitive complexity perceived by developers remains an open question. To mitigate this risk, we conduct a comprehensive human study (refer to Section 6.1 for details), where professional software engineers directly evaluate LLM-generated solutions against baseline solutions. The high agreement rates between human judgments and proxy metrics (94.6% for complexity and 96.0% for conciseness) provide empirical evidence that our chosen metrics reliably reflect genuine human preferences.

7 Related Work

7.1 Large Language Models for Code

LLMs have shown strong capabilities in code generation due to their large-scale training on diverse datasets, such as CodeLlama [34], WizardCoder [21], and DeepSeek-Coder [13]. These models are typically enhanced through instruction SFT to further optimize their code generation performance.

Given the challenge of collecting high-quality instructional data, researchers have increasingly adopted self-instruct methodologies to generate synthetic training data using powerful models like GPT-4 [4, 40, 44]. Evol-Instruct [21] employs sophisticated prompting strategies to generate more complex and diverse training instances. OSS-Instruct [45] enables LLMs to leverage real-world open-source code snippets, thereby improving the practical relevance and quality of generated solutions. While SFT boosts code quality, its exclusive focus on correct examples limits its ability to teach preference discrimination, as models never encounter negative examples [15].

7.2 Preference Optimization for Code Models

Recent researchers have employed Direct Preference Optimization (DPO) [32] to align models using pairwise preference data. DPO enables models to learn ranking preferences and select superior solutions (e.g., more efficient code) [30, 46, 52, 53]. Several studies have explored preference optimization specifically for code generation. Code-Optimize [11] constructs its training dataset from the MBPP-train subset, which comprises a limited set of 384 programming problems. PLUM [52] leverages GPT-4 to generate comprehensive test cases for validating and ranking code solutions, currently achieving state-of-the-art performance in preference optimization for code models. CodeDPO [53] uses a self-generation and validation mechanism to create balanced preference pairs, aiming to optimize both correctness and efficiency.

While both SFT and DPO have demonstrated success in natural language processing tasks, their effectiveness across diverse code preference types remains under-explored. Code preferences differ fundamentally from natural language preferences, requiring objective metrics such as correctness, efficiency, and security. Given their distinct optimization behaviors, SFT and DPO may serve complementary roles depending on the specific preference type, which complicates the selection of optimal training strategies for different code quality objectives.

8 Conclusion and Future Work

This paper systematically investigates the roles of SFT and DPO in aligning LLMs with diverse code preferences. Through theoretical analysis and empirical evidence, we demonstrate that SFT excels in types with objectively verifiable optimal solutions, while S&D enables superior exploration in types without objectively verifiable optimal solutions. Based on these insights, we propose **A**daptive **P**reference **O**ptimization (APO), a unified framework that dynamically integrates SFT and DPO strengths without requiring manual strategy selection. Extensive experiments across six representative code preference tasks validate our hypotheses and show that APO consistently matches or surpasses existing approaches while simplifying the training pipeline. Our work provides both theoretical foundations and practical guidance for code preference alignment. Future work will explore multi-turn programming scenarios and real-time human-in-the-loop alignment settings.

9 Data Availability

The replication of this paper is publicly available [3].

Acknowledgements

This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198 (SD2)), the National Natural Science Foundation of China (No.62572429), Ningbo Global in Innovation Center, Zhejiang University, the Fundamental Research Funds for the Central Universities (No.226-202200064), Zhejiang Provincial Natural Science Foundation of China (No.LY24F020008), the Key Research and Development Program of Zhejiang Province (No.2021C01105), the State Street Zhejiang University Technology Center.

References

[1] 2023. Hugging Face. https://huggingface.co

[2] 2025. The Radon library. https://github.com/rubik/radon

[3] 2026. Replication. https://github.com/vinci-grape/APO

[4] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] William G Cochran. 1977. Sampling techniques. *Johan Wiley & Sons Inc* (1977).

[7] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[8] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[9] Thomas Durieux. 2024. Empirical study of the docker smells impact on the image size. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[10] Yanjie Gao, Ruiming Lu, Haoxiang Lin, and Yueguo Chen. 2025. An Empirical Study of Issues in Large Language Model Training Systems. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 122–133.

[11] Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2025. Code-optimise: Self-generated preference data for correctness and efficiency. In *Findings of the Association for Computational Linguistics: NAACL 2025*. 79–94.

[12] Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. 2025. Repotransagent: Multi-agent llm framework for repository-aware code translation. *arXiv preprint arXiv:2508.17720* (2025).

[13] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[14] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. [n. d.]. Measuring Coding Challenge Competence With APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[15] Jiwoo Hong, Noah Lee, and James Thorne. 2024. ORPO: Monolithic Preference Optimization without Reference Model. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 11170–11189.

[16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[17] Jai Kannan, Scott Barnett, Luís Cruz, Anj Simmons, and Akash Agarwal. 2022. Mlsmellhound: A context-aware code analysis tool. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 66–70.

[18] Saeed Khaki, JinJin Li, Lan Ma, Liu Yang, and Prathap Ramachandra. 2024. RS-DPO: A Hybrid Rejection Sampling and Direct Preference Optimization Method for Alignment of Large Language Models. In *Findings of the Association for Computational Linguistics: NAACL 2024*. 1665–1680.

[19] Page Lawrence. 1998. The pagerank citation ranking: Bringing order to the web. *Tech Rep* (1998).

[20] Dianshu Liao, Xin Yin, Shidong Pan, Chao Ni, Zhenchang Xing, and Xiaoyu Sun. 2025. Navigating the Labyrinth: Path-Sensitive Unit Test Generation with Large Language Models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 687–699.

[21] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=UnUwSIgK5W

[22] AI Meta. 2024. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. *Meta AI Blog. Retrieved December* 20 (2024), 2024.

[23] Delano Oliveira, Reydne Santos, Benedito de Oliveira, Martin Monperrus, Fernando Castor, and Fernanda Madeiral. 2025. Understanding Code Understandability Improvements in Code Reviews. *IEEE Transactions on Software Engineering* 51, 1 (2025), 14–37.

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

[25] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced

by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[26] Yu Pan, Zhongze Cai, Huaiyang Zhong, Guanting Chen, and Chonghuan Wang. [n. d.]. What Matters in Data for DPO?. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[28] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. Coffe: A code efficiency benchmark for code generation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 242–265.

[29] Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Hao Zhang, Chenhao Ying, and Yuan Luo. 2025. SolEval: Benchmarking Large Language Models for Repository-level Solidity Smart Contract Generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 4388–4411.

[30] Zhiyuan Peng, Xin Yin, Zijie Zhou, Chenhao Ying, Chao Ni, and Yuan Luo. 2025. PrefGen: A Preference-Driven Methodology for Secure Yet Gas-Efficient Smart Contract Generation. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 317–329.

[31] Rafael Rafailov, Joey Hejna, Ryan Park, and Chelsea Finn. 2024. From r to Q^*: Your Language Model is Secretly a Q-Function. In *First Conference on Language Modeling*. https://openreview.net/forum?id=kEVcNxtqXk

[32] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2024).

[33] Yi Ren and Danica J. Sutherland. 2025. Learning Dynamics of LLM Finetuning. In *The Thirteenth International Conference on Learning Representations*. https://openrevie'w.net/forum?id=tPNHOoZFl9

[34] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[35] Antu Saha and Oscar Chaparro. 2025. Decoding the Issue Resolution Process in Practice via Issue Report Analysis: a Case Study of Firefox. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2316–2328.

[36] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[37] Matt Stuchlik, Bruno P. Kinoshita, and Donald Lee. 2025. The Cirron library. https://github.com/s7nfo/Cirron

[38] Zarrin Tasnim Sworna, Chadni Islam, and Muhammad Ali Babar. 2023. Apiro: A framework for automated security tools api recommendation. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–42.

[39] Fahim Tajwar, Anikait Singh, Archit Sharma, Rafael Rafailov, Jeff Schneider, Tengyang Xie, Stefano Ermon, Chelsea Finn, and Aviral Kumar. 2024. Preference fine-tuning of LLMs should leverage suboptimal, on-policy data. In *Proceedings of the 41st International Conference on Machine Learning*. 47441–47474.

[40] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[41] Bart Van Oort, Luís Cruz, Maurício Aniche, and Arie Van Deursen. 2021. The prevalence of code smells in machine learning projects. In *2021 IEEE/ACM 1st workshop on AI engineering-software engineering for AI (WAIN)*. IEEE, 1–8.

[42] Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, et al. 2024. Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models. *arXiv preprint arXiv:2408.01605* (2024).

[43] Simin Wang, Liguo Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2022. Machine/deep learning for software engineering: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1188–1231.

[44] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

[45] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *International Conference on Machine Learning*. PMLR, 52632–52657.

[46] 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 *International Conference on Machine Learning*. PMLR, 54983–54998.

[47] Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. 2025. ProSec: Fortifying Code LLMs with Proactive Security Alignment. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=Ym19zWky7W
[48] Boyang Yang, Luyao Ren, Xin Yin, Jiadong Ren, Haoye Tian, and Shunfu Jin. 2025. Input reduction enhanced llm-based program repair. *arXiv preprint arXiv:2507.15251* (2025).
[49] Xin Yin, Chao Ni, and Shaohua Wang. 2024. Multitask-based Evaluation of Open-Source LLM on Software Vulnerability. *IEEE Transactions on Software Engineering* 01 (2024), 1–16.
[50] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.
[51] Xin Yin, Chao Ni, Xiaodan Xu, and Xiaohu Yang. 2025. What You See Is What You Get: Attention-based Self-guided Automatic Unit Test Generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*.
[52] Dylan Zhang, Shizhe Diao, Xueyan Zou, and Hao Peng. 2024. Plum: Preference learning plus test cases yields better code language models. *arXiv e-prints* (2024), arXiv preprint arXiv:2406.06887.
[53] 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. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15854–15871.
[54] 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. In *Findings of the Association for Computational Linguistics: ACL 2025*. 9578–9591.
[55] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. 2023. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792* (2023).

Received 2025-09-08; accepted 2026-03-24