

CoEffi: Enhance Efficient Code Generation via Dynamic Collaborative Decoding

Jiahui Lin¹, Jingrui Zhang¹, Zixuan Zhang¹, Meilin Li¹,
Man Fu¹, Siyao Yu¹, Weifeng Sun^{2,*}, Meng Yan^{1,*}
¹Chongqing University ²Singapore Management University

jiahui@stu.cqu.edu.cn

*Corresponding authors: wfsun@smu.edu.sg, mengy@ccqu.edu.cn

Abstract

Code LLMs exhibit remarkable capabilities in automated code generation, but the generated code often suffers from low computational efficiency. Existing solutions have two limitations: (1) direct fine-tuning lacks transferability across model scales; (2) The efficiency optimization method for iterative generation-evaluation loops incurs high overhead. We propose CoEffi, which adopts inference-side intervention that directly operates on the decoding process without modifying base model parameters, enabling real-time efficiency improvement across different model scales while avoiding iterative reevaluation. CoEffi uses a small efficiency expert model for proposal and the base model to evaluate its acceptability, additionally introducing a region-aware dynamic acceptance threshold mechanism that maintains a sliding window to record historical acceptance rates and adjusts the dynamic threshold accordingly. We extensively evaluate CoEffi on four mainstream code LLM families and three efficiency benchmarks. Compared with vanilla instruction-tuned models, CoEffi improves average Beyond@1 by 9.12%–38.62% across the evaluated model families. It also improves Pass@1 on Mercury and ENAMEL, while EffiBench exhibits model-dependent correctness gains and regressions. A family specific trained expert can be reused across compatible base model scales within the same model family.

1 Introduction

In recent years, modern Large Language Models (LLMs), such as GPT-4o (Hurst et al., 2024), Llama 3 (Grattafiori et al., 2024), and Gemini 2.5 (Comanici et al., 2025), have advanced rapidly, demonstrating significant progress in natural language processing. Building upon this foundation, code LLMs, including DeepSeek-Coder (Guo et al., 2024), Qwen2.5-Coder (Hui et al., 2024), OpenCoder (Huang et al., 2025), and StarCoder2

(Lozhkov et al., 2024), have been trained on massive code corpora and code-related natural language texts. These models demonstrate strong capabilities in synthesizing syntactically correct and functionally complete programs, changing development practices. According to the 2025 *Stack Overflow Developer Survey* (Stack Overflow, 2025), a majority of professional developers now utilize AI tools on a daily basis.

However, while these models achieve near-human performance on traditional benchmarks (e.g., HumanEval, Chen et al., 2021; MBPP, Austin et al., 2021; and EvalPlus, Liu et al., 2023), current evaluation paradigms focus almost exclusively on the functional correctness of the generated code. The actual performance of these models in terms of computational efficiency remains under-explored. For instance, as illustrated in Figure 1, DeepSeek-Coder-6.7B-Instruct generates a functionally correct but highly inefficient solution. It results in a time complexity of $\mathcal{O}(n \times L)$. In response to this demand, recent benchmarks such as Mercury (Du et al., 2024), ENAMEL (Qiu et al., 2025), and EffiBench (Huang et al., 2024b) have introduced specialized efficiency metrics. Although these specialized benchmarks successfully complement traditional ones, existing methods for generating efficient code also have limitations.

Recent efforts to improve code efficiency primarily follow two directions: directly fine-tuning code LLMs for efficiency awareness (e.g., EffiCoder, Huang et al., 2024c; CodeDPO, Zhang et al., 2025a; SkelDPO, Yu and Lyu, 2026) or performing post-generation self-optimization based on execution feedback (e.g., EffiLearner, Huang et al., 2024a; Afterburner, Du et al., 2026). However, these approaches have two key limitations. The first limitation lies in the lack of transferability within a model family. Fine-tuning-based methods produce weight modifications tied to a specific model instance. The second is the inefficient generation-evaluation loop

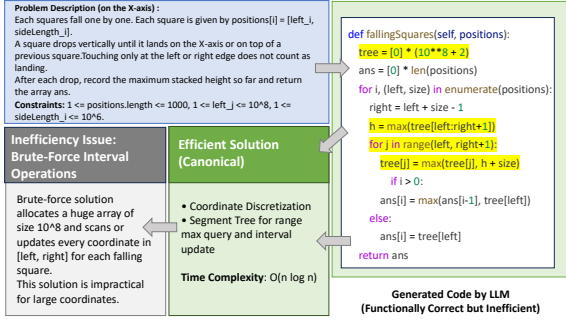


Figure 1: A motivational example comparing an LLM-generated brute-force solution with a traditional efficient segment tree solution.

inherent in post-generation optimization. These methods require repeatedly generating and evaluating code, wasting substantial computational resources before corrections take effect.

To address these limitations, we introduce CoEffi, a collaborative decoding framework designed for the generation process of code LLMs. CoEffi enhances efficiency by intervening in token selection in real-time through a decoupled large-small model synergy. CoEffi uses a smaller LoRA tuned model from the same family as an efficiency expert (Hu et al., 2021). Tokenizer and architectural compatibility enable token level collaboration, while base model evaluation and fallback sampling reduce intervention risk. Each model family requires its own trained expert, which can be reused across compatible base model scales within that family. Furthermore, to balance aggressive efficiency rewrites with the preservation of functional correctness, CoEffi incorporates a region-aware dynamic acceptance threshold mechanism that adaptively relaxes or tightens acceptance criteria based on historical decoding states.

To verify the generalization of our method, we selected four model families (DeepSeek-Coder, Qwen2.5-Coder, OpenCoder, and StarCoder2), which have demonstrated effectiveness in code generation in previous studies and cover model sizes from 1.3B to 8B. Experiments across Mercury, ENAMEL, and EffiBench show consistent efficiency improvements across the evaluated model families. Pass@1 improves consistently on Mercury and ENAMEL, while EffiBench presents model dependent gains and regressions. Notably, our framework exhibits even stronger potential in high difficulty programming tasks and maintains robustness across various sampling configurations.

The primary contributions of this paper are sum-

marized as follows:

- We propose CoEffi, a collaborative decoding framework integrating large-small model synergy and a region-aware dynamic acceptance threshold mechanism, which has not been explored for efficient code generation.
- We evaluate CoEffi on four code LLM families and three efficiency centric benchmarks. CoEffi consistently improves efficiency metrics across the evaluated settings and improves functional correctness on Mercury and ENAMEL, while EffiBench shows model dependent correctness trade-offs.
- To support the open science community, we have open-sourced our framework and experimental scripts for future research.¹

2 Related Work

2.1 Collaborative Decoding

The core idea of collaborative decoding is to leverage the complementary advantages of a large language model and lightweight auxiliary models, thereby enhancing overall multi-domain expertise, safety, or attribute control without retraining the large model. Recent studies (Shen et al., 2024; Cao et al., 2023; Liu et al., 2021; Zhong et al., 2023) modify the token distribution of the large model through simple probability operations—such as products, ratios, or distribution reconstruction—thereby enhancing specific capabilities while maintaining fluency and minimizing inference overhead.

We emphasize that although CoEffi is also an implementation based on this paradigm, we introduce two innovations for enhancing the efficiency of code generated by code LLMs based on two insights: (1) *macro continuity*: forming efficient algorithmic logic often spans a long code generation process, which requires continuous intervention to satisfy long-range dependencies (Sun et al., 2020; Zhang et al., 2025b; Mukherjee et al., 2021). Therefore, we adopt a dynamic threshold; when the efficiency expert model generates highly consistent and efficient code, the dynamic threshold is relaxed to achieve more frequent interventions. (2) *micro explorability*: efficient algorithms often have multiple implementations, requiring the model to possess the ability to explore diverse structures (Lee

¹<https://github.com/qingq-l/CoEffi>

et al., 2025; Dilhara et al., 2024). Therefore, we employ random sampling scaled by the dynamic threshold, increasing generation diversity to explore more efficient patterns.

2.2 Methods for Enhancing Code Efficiency

Existing methods can be classified by the timing of intervention. *Training phase*: EffiCoder (Huang et al., 2024c) utilizes execution feedback to construct efficiency datasets for fine-tuning; RLEF (Gehring et al., 2024) adopts reinforcement learning for iterative optimization; CodeDPO (Zhang et al., 2025a) automatically constructs preference pairs through execution verification for DPO alignment; SkelDPO (Yu and Lyu, 2026) further extracts program skeletons, incorporating both the original code and the skeletons into DPO optimization; and ReflexiCoder (Jiang et al., 2026) internalizes the efficiency reflection chain into model weights.

Pre-generation phase: LLM4EFFI (Ye et al., 2025) first generates an efficiency analysis report to guide subsequent code generation. Unlike post-processing methods that rely on execution feedback, it attempts to prevent inefficient code by targeting generation goals from the outset. We consider this as another solution to Limitation 2, but since it operates in the pre-generation phase and does not involve model fine-tuning, its paradigm differs from that of CoEffi. Therefore, it is not included in the experimental comparison.

Post-generation phase: ORPS (Yu et al., 2024) selects efficient candidates via tree search; EffiLearner (Huang et al., 2024a) and Afterburner (Du et al., 2026) iteratively rewrite code based on execution feedback. Compared to the above methods, our CoEffi fills the gap in intervention timing by intervening during the generation process. Although post-generation methods follow a different computational workflow, we additionally compare CoEffi with EffiLearner under the same local-model configurations to provide a practical reference.

3 Methodology

In this section, we elaborate on the specific implementation of CoEffi. As shown in Figure 2, the CoEffi framework primarily consists of two core modules: (A) efficiency expert model fine-tuning (Section 3.1), which obtains an efficiency expert model via parameter efficient fine-tuning; (B) dynamic collaborative decoding (Section 3.2), which

uses a dynamic threshold for large-small model synergistic generation.

3.1 Efficiency Expert Model Fine-Tuning

3.1.1 Motivation for Parameter Efficient Fine-Tuning

In CoEffi, the efficiency expert model is the source of code efficiency. CoEffi introduces efficient tokens by having the efficiency expert model intervene in the base model during the decoding process. To obtain the efficiency expert model, we adopt LoRA (Hu et al., 2021) for parameter efficient fine-tuning of the code LLM, due to the high acquisition cost and limited availability of efficient code optimization data, as well as the possibility that users may need to deploy the model in resource constrained private environments. This approach reduces the required scale of training data and also alleviates the memory and computational burden during deployment. Compared with fine-tuning methods tied to a specific model instance, CoEffi allows one family specific expert to support compatible base model scales within the same family.

3.1.2 Skeleton Augmented Preference Optimization

SkelDPO (Yu and Lyu, 2026) extends standard DPO by augmenting the paired “inefficient-efficient” code samples with skeleton pairs extracted from the code, i.e., structural abstractions that reflect high-performance implementation patterns. The training objective is a convex combination of a code preference loss $\mathcal{L}_{\text{code}}$ and a skeleton preference loss $\mathcal{L}_{\text{skel}}$:

$$\mathcal{L} = \alpha \mathcal{L}_{\text{code}} + (1 - \alpha) \mathcal{L}_{\text{skel}}, \quad \alpha \in [0, 1] \quad (1)$$

During training, the model takes as input tokenized prompt-response pairs (code or skeleton) with attention and response masks, and outputs the scalar loss $\mathcal{L}$ defined in Equation (1). Specifically, each sample consists of a problem description q , efficient code c_w , inefficient code c_l , and their skeletons s_w, s_l . The code branch concatenates q with c_w (or c_l) to form input sequences, while the skeleton branch concatenates q with s_w (or s_l) and uses a response mask to mark the skeleton answer part. Both the trainable model (with LoRA) and a frozen reference model (without LoRA) compute the token-average log probability $\bar{p}$ for each input sequence, yielding the DPO reward $r = \beta(\bar{p}_\theta - \bar{p}_{\text{ref}})$. The code loss and skeleton loss

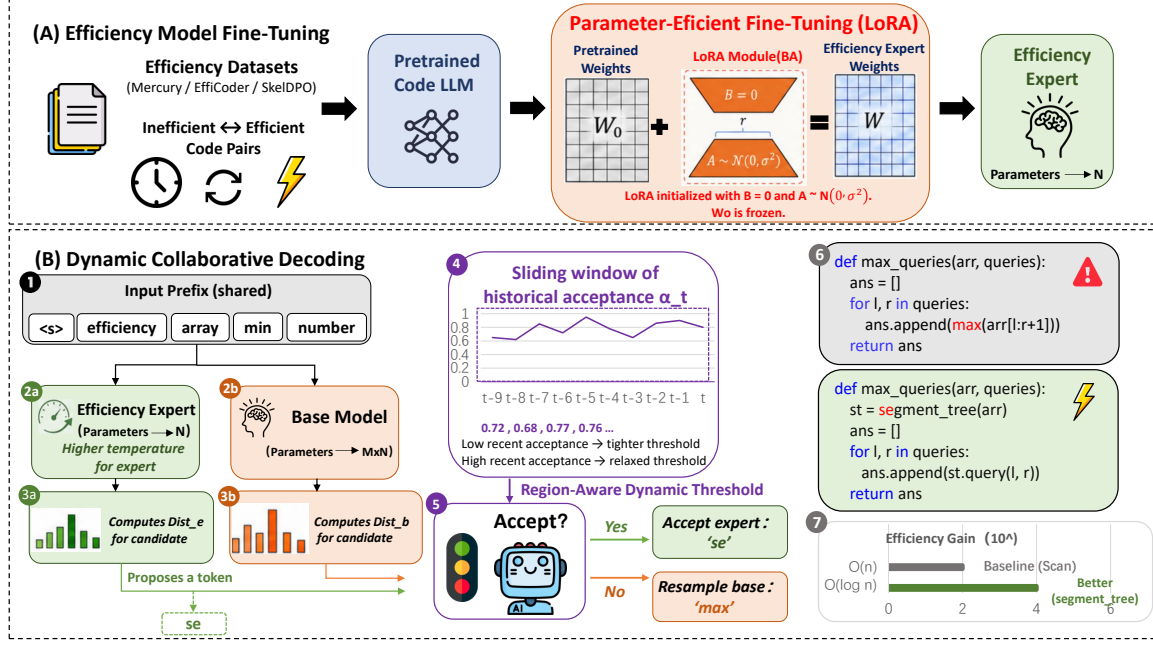


Figure 2: Overview of the CoEffi framework. (A) Efficiency expert model fine-tuning via LoRA. (B) Dynamic collaborative decoding with a region-aware dynamic threshold.

each adopt the form $-\log \sigma(r_w - r_l)$. For a detailed definition of the loss formula, please refer to Appendix A. Details regarding the comparative experiments among different training paradigms and the configuration of hyperparameters are provided in Appendix C.4.

3.2 Dynamic Collaborative Decoding

CoEffi’s dynamic collaborative decoding is built upon two key observations.

First, the asymmetry of expert proposals: the efficiency tuned model exhibits a higher tendency to propose efficient rewrites at its positions, but its ability to guarantee functional correctness, especially satisfying long range dependencies and maintaining overall code integrity, remains weaker than that of the base model. Hence the expert is suitable as a proposer, while the base model must act as the evaluator and decision maker.

Second, the local stationarity of model disagreement: the degree of agreement between the two models, captured by the ratio r_t (Equation 3), tends to be locally smooth. If the base model consistently accepts the expert’s proposals for several consecutive steps, it is likely to continue doing so in the near future; similarly, consecutive rejections often indicate a region where the expert is unreliable. This property enables us to use a sliding window of recent agreement history to estimate the

current risk level and adapt the dynamic threshold accordingly.

Based on these observations, we adopt a **decoupled evaluation** framework. At each time step, the expert proposes a candidate token, the base model evaluates its acceptance degree, and we adaptively adjust the acceptance criteria according to the recent history. The process proceeds as follows.

Candidate proposal At time step t , we sample a candidate token $\tilde{x}_t$ from the expert’s distribution:

$$\tilde{x}_t \sim p_E(\cdot \mid x_1, \dots, x_{n+t-1}) \quad (2)$$

A slightly higher temperature is used for the expert to encourage exploration of aggressive efficiency rewrites.

Acceptance evaluation To quantify how much the base model accepts this candidate, we compute the conditional probability ratio:

$$r_t = \frac{p_B(\tilde{x}_t \mid x_1, \dots, x_{n+t-1})}{p_E(\tilde{x}_t \mid x_1, \dots, x_{n+t-1}) + \epsilon} \quad (3)$$

The objective acceptance degree is then defined as $\alpha_t = \min(r_t, 1)$. Truncation at 1 prevents high-frequency tokens from dominating the scale.

Region-aware dynamic acceptance threshold (for macro continuity). Based on the above understanding of **macro continuity**, we adopt a dynamic threshold instead of a fixed one. Specifically,

we maintain a sliding window H_t of length w that records the recent α_j , and compute their mean:

$$\bar{\alpha}_t = \frac{1}{w} \sum_{j=t-w+1}^t \alpha_j \quad (4)$$

$\bar{\alpha}_t$ serves as an online estimate of the current risk level, leveraging the local stationarity of disagreement.

- **Penalty mechanism:** when $\bar{\alpha}_t < 0.3$ (consecutive rejections $\rightarrow$ high risk region), we increase the dynamic threshold to make acceptance stricter, preventing aggressive proposals from damaging functional correctness.
- **Reward mechanism:** when $\bar{\alpha}_t > 0.8$ (consecutive acceptances $\rightarrow$ low risk region), we decrease the dynamic threshold to accept more efficient interventions.

The dynamic threshold τ_t is updated as:

$$\begin{aligned} \tau_{t+1} &= \text{clip}(\tau_t + \Delta_t, \tau_{\min}, \tau_{\max}), \\ \Delta_t &= \begin{cases} +\delta, & \bar{\alpha}_t < 0.3 \\ -\delta, & \bar{\alpha}_t > 0.8 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \quad (5)$$

Hard boundaries $[\tau_{\min}, \tau_{\max}]$ prevent two undesirable extremes: an unbounded increase would cause all expert candidates to be rejected, while an unbounded decrease would cause all expert candidates to be accepted regardless of quality.

Random acceptance sampling (for micro explorability). Based on the above understanding of **micro explorability**, we employ random acceptance sampling where the dynamic threshold τ_t acts as a scaling factor, rather than a deterministic cutoff. We draw a uniform random number $u_t \sim \text{Uniform}(0, 1)$ and scale it by the dynamic threshold:

$$\tilde{u}_t = u_t \cdot \tau_t \quad (6)$$

The final token is decided by:

$$x_{n+t} = \begin{cases} \tilde{x}_t, & \text{if } \tilde{u}_t < \min(1, r_t) \\ \hat{x}_t \sim p_B(\cdot \mid x_{<n+t}), & \text{otherwise} \end{cases} \quad (7)$$

That is, if the scaled random number $\tilde{u}_t$ is less than the minimum acceptance probability $\min(1, r_t)$, we **accept** the expert’s candidate token $\tilde{x}_t$; otherwise, we **reject** the expert’s proposal and **fallback** to sampling a token $\hat{x}_t$ directly from the base

model’s distribution p_B . This fallback mechanism ensures functional correctness when the expert’s proposal is too aggressive or unreliable.

When $\tau_t = 1$, this reduces to standard speculative acceptance. When $\tau_t > 1$ (conservative state), acceptance is harder; when $\tau_t < 1$ (trusting state), acceptance is easier. This multiplicative coupling unifies region judgment (from $\bar{\alpha}_t$) and step wise divergence (from r_t) into a single smooth acceptance mechanism, achieving both macro continuity and micro explorability. Default hyperparameters are: $\tau_0 = 0.5$, $w = 10$, $\delta = 0.10$, $\tau_{\min} = 0.3$, $\tau_{\max} = 1.2$.

4 Experimental Setup

In this section, the experimental setup will be introduced. We also provide a summary table of some key hyperparameters. Please refer to Appendix B.

4.1 Datasets and Benchmarks

We use three efficiency-oriented Python training sets to construct baselines and candidate efficiency expert models for collaborative decoding: Mercury (Du et al., 2024), EffiCoder (Huang et al., 2024c), and SkelDPO (Yu and Lyu, 2026). Evaluation is conducted on Mercury-eval (Du et al., 2024), ENAMEL (Qiu et al., 2025), and EffiBench (Huang et al., 2024b). Dataset construction details, benchmark specifications, and contamination controls are provided in Appendix C.1, C.2, and C.3.

4.2 Models and Baselines

We employ four instruction-tuned code LLMs as base models: DeepSeek-Coder (6.7B) (Guo et al., 2024), Qwen2.5-Coder (7B) (Hui et al., 2024), OpenCoder (8B) (Huang et al., 2025), and StarCoder2 (7B) (Lozhkov et al., 2024). Baselines include the original instruction tuned model, SFT, DPO, and SkelDPO. We additionally compare CoEffi with EffiLearner on Mercury under the same local model configurations. For collaborative decoding, each base model is paired with a smaller same-family expert (DeepSeek-Coder-1.3B, Qwen2.5-Coder-1.5B, OpenCoder-1.5B, and StarCoder2-3B) fine-tuned with LoRA. Training configurations are provided in Appendix C.4.

4.3 Collaborative Inference

CoEffi operates at generation time without evaluation-feedback loops, attaching a lightweight efficiency expert model to the base model via LoRA

Table 1: Comparison of code efficiency across benchmarks.

Model	Variant	Beyond@1 ↑	Eff@1 ↑	ET (s) ↓	NET ↓	MU (MB) ↓	NMU ↓	TMU (MB-s) ↓	NTMU ↓
DeepSeek-Coder (6.7B)	Instruct	0.5985	0.2208	0.24	2.66	37.02	1.10	16.07	3.65
	DPO	0.6252 (↑4.5%)	0.2552 (↑15.6%)	0.22 (↓8.3%)	2.41 (↓9.4%)	63.30 (↑71.0%)	1.80 (↑63.6%)	78.14 (↑386.2%)	14.58 (↑299.5%)
	SFT	0.6412 (↑7.1%)	0.4347 (↑96.9%)	0.21 (↓12.5%)	2.42 (↓9.0%)	65.48 (↑76.9%)	1.85 (↑68.2%)	81.65 (↑408.1%)	16.16 (↑342.7%)
	SkelDPO	0.6292 (↑5.1%)	0.2884 (↑30.6%)	0.21 (↓12.5%)	2.51 (↓5.6%)	68.22 (↑84.3%)	1.94 (↑76.4%)	122.44 (↑661.9%)	25.21 (↑590.7%)
	CoEffi	0.6692 (↑11.8%)	0.3072 (↑39.1%)	0.20 (↓16.7%)	2.15 (↓19.2%)	36.91 (↓0.3%)	1.07 (↓2.7%)	13.31 (↓17.2%)	2.89 (↓20.8%)
DeepSeek-Coder (1.3B)	Instruct	0.4183	0.2138	0.44	4.71	113.75	2.94	205.81	30.61
	CoEffi	0.4436 (↑6.0%)	0.2950 (↑38.0%)	0.37 (↓15.9%)	3.88 (↓17.6%)	47.49 (↓58.3%)	1.24 (↓57.8%)	39.48 (↓80.8%)	6.07 (↓80.2%)
Qwen2.5-Coder (7B)	Instruct	0.6656	0.1760	0.18	2.04	33.68	1.00	7.08	1.56
	DPO	0.6908 (↑3.8%)	0.1797 (↑2.1%)	0.16 (↓11.1%)	1.85 (↓9.3%)	38.31 (↑13.7%)	1.12 (↑12.0%)	17.17 (↑142.5%)	3.83 (↑145.5%)
	SFT	0.6832 (↑2.6%)	0.3922 (↑122.8%)	0.16 (↓11.1%)	1.91 (↓6.4%)	34.74 (↑3.1%)	1.00 (↑0.0%)	7.07 (↓0.1%)	1.49 (↓4.5%)
	SkelDPO	0.6842 (↑2.8%)	0.2109 (↑19.8%)	0.17 (↓5.6%)	2.02 (↓1.0%)	61.60 (↑82.9%)	1.82 (↑82.0%)	75.47 (↑966.0%)	17.11 (↑996.8%)
	CoEffi	0.7263 (↑9.1%)	0.3156 (↑79.3%)	0.15 (↓16.7%)	1.78 (↓12.7%)	33.66 (↓0.1%)	1.01 (↑1.0%)	6.90 (↓2.5%)	1.55 (↓0.6%)
Qwen2.5-Coder (3B)	Instruct	0.5779	0.1758	0.28	2.99	39.30	1.10	21.11	3.76
	CoEffi	0.6091 (↑5.4%)	0.2161 (↑22.9%)	0.25 (↓10.7%)	2.67 (↓10.7%)	63.54 (↑61.7%)	1.87 (↑70.0%)	80.60 (↑281.8%)	16.88 (↑348.9%)
Qwen2.5-Coder (1.5B)	Instruct	0.3470	0.1637	0.49	5.35	39.74	1.05	21.82	3.55
	CoEffi	0.5413 (↑56.0%)	0.3752 (↑129.2%)	0.35 (↓28.6%)	3.58 (↓33.1%)	54.52 (↑37.2%)	1.31 (↑24.8%)	45.84 (↑110.1%)	5.97 (↑68.2%)
OpenCoder (8B)	Instruct	0.5129	0.2661	0.19	2.26	40.74	1.21	24.42	5.59
	DPO	0.6359 (↑24.0%)	0.2923 (↑9.8%)	0.16 (↓15.8%)	1.90 (↓15.9%)	37.11 (↓8.9%)	1.13 (↓6.6%)	19.88 (↓18.6%)	4.96 (↓11.3%)
	SFT	0.6706 (↑30.7%)	0.4286 (↑61.1%)	0.15 (↓21.1%)	1.62 (↓28.3%)	35.04 (↓14.0%)	1.02 (↓15.7%)	7.08 (↓71.0%)	1.40 (↓75.0%)
	SkelDPO	0.6882 (↑34.2%)	0.2875 (↑8.0%)	0.18 (↓5.3%)	2.14 (↓5.3%)	36.03 (↓11.6%)	1.06 (↓12.4%)	14.25 (↓41.6%)	3.18 (↓43.1%)
	CoEffi	0.7110 (↑38.6%)	0.3010 (↑13.1%)	0.15 (↓21.1%)	1.82 (↓19.5%)	33.55 (↓17.6%)	1.00 (↓17.4%)	8.48 (↓65.3%)	2.02 (↓63.9%)
OpenCoder (1.5B)	Instruct	0.5005	0.1889	0.22	2.38	50.13	1.35	36.03	6.37
	CoEffi	0.6446 (↑28.8%)	0.2607 (↑38.0%)	0.20 (↓9.1%)	2.14 (↓10.1%)	36.28 (↓27.6%)	0.99 (↓26.7%)	8.92 (↓75.2%)	1.48 (↓76.8%)
StarCoder2 (7B)	Instruct	0.3744	0.2123	0.57	5.70	152.62	3.40	384.77	44.01
	DPO	0.3947 (↑5.4%)	0.2507 (↑18.1%)	0.34 (↓40.4%)	3.91 (↓31.4%)	101.14 (↓33.7%)	2.74 (↓19.4%)	222.72 (↓42.1%)	39.90 (↓9.3%)
	SFT	0.3824 (↓2.1%)	0.3614 (↑70.2%)	0.47 (↓17.5%)	4.70 (↓17.5%)	252.08 (↑65.2%)	5.48 (↑61.2%)	669.97 (↑74.1%)	65.53 (↑48.9%)
	SkelDPO	0.3822 (↓2.1%)	0.2159 (↑1.7%)	0.50 (↓12.3%)	4.80 (↓15.8%)	231.48 (↑51.7%)	5.00 (↑47.1%)	558.21 (↑45.1%)	60.56 (↑37.6%)
	CoEffi	0.4374 (↑16.8%)	0.2970 (↑39.9%)	0.49 (↓14.0%)	5.01 (↓12.1%)	78.43 (↓48.6%)	1.83 (↓46.2%)	102.64 (↓73.3%)	13.75 (↓68.8%)
StarCoder2 (3B)	Instruct	0.1806	0.2039	0.84	8.29	81.17	2.17	176.60	30.13
	CoEffi	0.2098 (↑16.2%)	0.2656 (↑30.3%)	0.66 (↓21.4%)	7.10 (↓14.4%)	80.37 (↓1.0%)	2.04 (↓6.0%)	156.79 (↓11.2%)	26.38 (↓12.5%)

adapters. The base model and expert operate under distinct sampling configurations coupled with a dynamic token acceptance mechanism. Complete inference configurations are provided in Appendix C.5.

4.4 Evaluation Metrics

We report Pass@1 for functional correctness, measuring the fraction of tasks that pass all tests.

For efficiency, we adopt Beyond@1 on Mercury (Du et al., 2024), which quantifies the runtime improvement of generated code against a set of reference solutions. Failed solutions receive a score of 0. We also report Eff@1 on ENAMEL (Qiu et al., 2025), defined as the average efficiency score against the expert reference. (Both metrics: higher is better). On EffiBench (Huang et al., 2024b), we report six resource-efficiency metrics: Execution Time (ET), Normalized ET (NET), Max Memory Usage (MU), Normalized MU (NMU), Total Memory Usage (TMU), and Normalized TMU (NTMU). NET, NMU, and NTMU are ratios to the canonical solution; lower is better for all six metrics. Formal definitions are provided in Appendix C.6.

5 Main Results and Analysis

5.1 Overall Performance on Efficiency

We evaluate CoEffi on four code LLMs, comparing it against the vanilla instruction-tuned models and fine-tuning-based approaches (DPO, SFT,

SkelDPO) across three efficiency-centric benchmarks. As shown in Table 1, CoEffi achieves a balanced optimization of execution time and memory usage on most models.

Compared to the vanilla instruction-tuned models, CoEffi yields substantial efficiency gains across all benchmarks: Beyond@1 improves by 5%–56% on Mercury, Eff@1 improves by 13%–129% on ENAMEL, NET drops by 10%–33% on EffiBench, and NTMU shrinks by up to 69% on memory-intensive models (e.g., StarCoder2 7B). These results indicate that CoEffi achieves synergistic optimization of execution efficiency and memory footprint across benchmarks, whereas fine-tuning methods often suffer from a pronounced time–space imbalance.

Two patterns highlight CoEffi’s advantages. First, on Qwen2.5-Coder 7B under EffiBench, CoEffi reduces NET by 12.7% while NMU and NTMU change by +1.0% and −0.6%, respectively, achieving spatial efficiency close to human-written canonical solutions and realizing dual optimization of time and space efficiency. In contrast, SkelDPO-baseline inflates NTMU by nearly 10× (+996.8%). This difference is consistent with CoEffi’s design: although the expert learns efficiency patterns from SkelDPO data, the base model retains final decision authority and can filter proposals that are incompatible with the current context. Second, CoEffi generalizes well to smaller models (1.5B–3B), achieving, for example, a 56.0% Beyond@1 improvement on

Table 2: Comparison of functional correctness across benchmarks.

Model	Variant	pass@1 (Mercury) ↑	pass@1 (ENAMEL) ↑	pass@1 (EffiBench) ↑
DeepSeek-Coder (6.7B)	Instruct	0.7736	0.3450	0.4870
	DPO	0.8019 (↑ 3.7%)	0.3873 (↑ 12.3%)	0.4670 (↓ 4.1%)
	SFT	0.7877 (↑ 1.8%)	0.7183 (↑ 108.2%)	0.4640 (↓ 4.7%)
	SkelDPO	0.8019 (↑ 3.7%)	0.4295 (↑ 24.5%)	0.4810 (↓ 1.2%)
	CoEffi	0.8255 (↑ 6.7%)	0.5352 (↑ 55.1%)	0.4690 (↓ 3.7%)
DeepSeek-Coder (1.3B)	Instruct	0.5519	0.3170	0.2400
	CoEffi	0.5849 (↑ 6.0%)	0.4436 (↑ 39.9%)	0.2800 (↑ 16.7%)
Qwen2.5-Coder (7B)	Instruct	0.8208	0.2464	0.4930
	DPO	0.8538 (↑ 4.0%)	0.2471 (↑ 0.3%)	0.5640 (↑ 14.4%)
	SFT	0.8443 (↑ 2.9%)	0.6126 (↑ 148.6%)	0.5540 (↑ 12.4%)
	SkelDPO	0.8396 (↑ 2.3%)	0.2887 (↑ 17.2%)	0.5890 (↑ 19.5%)
	CoEffi	0.8821 (↑ 7.5%)	0.4577 (↑ 85.8%)	0.5760 (↑ 16.8%)
Qwen2.5-Coder (3B)	Instruct	0.7311	0.2676	0.4460
	CoEffi	0.7642 (↑ 4.5%)	0.3521 (↑ 31.6%)	0.4780 (↑ 7.2%)
Qwen2.5-Coder (1.5B)	Instruct	0.4575	0.3028	0.2950
	CoEffi	0.6792 (↑ 48.5%)	0.5985 (↑ 97.7%)	0.2720 (↓ 7.8%)
OpenCoder (8B)	Instruct	0.6462	0.4436	0.5570
	DPO	0.7689 (↑ 19.0%)	0.4507 (↑ 1.6%)	0.5420 (↓ 2.7%)
	SFT	0.8066 (↑ 24.8%)	0.7535 (↑ 69.9%)	0.4490 (↓ 19.4%)
	SkelDPO	0.8538 (↑ 32.1%)	0.4929 (↑ 11.1%)	0.5580 (↑ 0.2%)
	CoEffi	0.8679 (↑ 34.3%)	0.4929 (↑ 11.1%)	0.4460 (↓ 19.9%)
OpenCoder (1.5B)	Instruct	0.6415	0.3521	0.3250
	CoEffi	0.8066 (↑ 25.7%)	0.4366 (↑ 24.0%)	0.3490 (↑ 7.4%)
StarCoder2 (7B)	Instruct	0.4906	0.3591	0.1620
	DPO	0.5039 (↑ 2.7%)	0.4154 (↑ 15.7%)	0.1490 (↓ 8.0%)
	SFT	0.5094 (↑ 3.8%)	0.6126 (↑ 70.6%)	0.1510 (↓ 6.8%)
	SkelDPO	0.5047 (↑ 2.9%)	0.4225 (↑ 17.7%)	0.1750 (↑ 8.0%)
	CoEffi	0.5708 (↑ 16.3%)	0.5070 (↑ 41.2%)	0.1870 (↑ 15.4%)
StarCoder2 (3B)	Instruct	0.2406	0.3570	0.1530
	CoEffi	0.3255 (↑ 35.3%)	0.4507 (↑ 26.3%)	0.1520 (↓ 0.7%)

Qwen2.5-Coder 1.5B and a 30.3% Eff@1 gain on StarCoder2 3B, demonstrating that CoEffi remains effective across different model scales.

In a few configurations, some fine-tuning methods outperform CoEffi. For example, on StarCoder2 7B, CoEffi underperforms other fine-tuning baselines in NET reduction, yet it performs well on Beyond@1 and Eff@1 (both reflecting temporal efficiency) and maintains bounded memory overhead. Overall, CoEffi improves execution efficiency across multiple dimensions, with model dependent trade-offs in memory use and inference cost.

Under the same local-model configurations on Mercury, CoEffi improves Beyond@1 over EffiLearner by 1.26–3.33 percentage points across all four models. On DeepSeek-Coder 6.7B, Qwen2.5-Coder 7B, and StarCoder2 7B, CoEffi also improves Pass@1 by 1.42–3.78 percentage points and reduces the recorded runtime by 9.28%–24.33%. On OpenCoder 8B, EffiLearner achieves 1.42 percentage points higher Pass@1, while CoEffi takes 20.22% longer and achieves 1.80 percentage points higher Beyond@1.

5.2 Effect on Functional Correctness

We investigate the impact of CoEffi on functional correctness (Pass@1) under the same configuration. Table 2 summarizes the results. On Mercury and ENAMEL, CoEffi consistently raises Pass@1. Relative gains range from 4.5% to 34.3% on Mercury and 11.1% to 97.7% on ENAMEL. This suggests that dynamic intervention can provide logical cor-

rections for conventional programming tasks.

On EffiBench, the experimental results present a mixed picture. CoEffi has different impacts on Pass@1 across different models, and similar mixed results also appear in the fine-tuning baselines. This indicates that in EffiBench, there is a more complex tradeoff between code efficiency improvement and functional correctness.

To understand why CoEffi improves Pass@1, we conducted a token-level analysis, tracking whether critical tokens originated from the expert model. For tasks where the vanilla instruction-tuned models failed but CoEffi succeeded, we investigated whether the critical shifts in algorithmic logic were directly driven by the expert’s interventions.

Using DeepSeek-Coder (6.7B) on the Mercury-eval benchmark as a case study, we identified 18 tasks that transitioned from incorrect to correct after applying CoEffi. In 17 of these cases, the tokens proposed by the efficiency expert played the decisive role in correcting the logic (the remaining one was also indirectly influenced by the intervention). Furthermore, experimental results consistently show that CoEffi outperforms both the standalone base model and the standalone expert model. For detailed case analyses of the 18 tasks and Pass@1 comparison results, see Appendix D.

These cases suggest that efficiency expert proposals can introduce alternative implementation patterns that improve correctness on some tasks. The base model evaluates each proposal against the complete generated prefix, and fallback sampling limits incompatible interventions. This mechanism does not guarantee correctness: as discussed in Appendix G, CoEffi can inherit systematic errors from its expert’s training data.

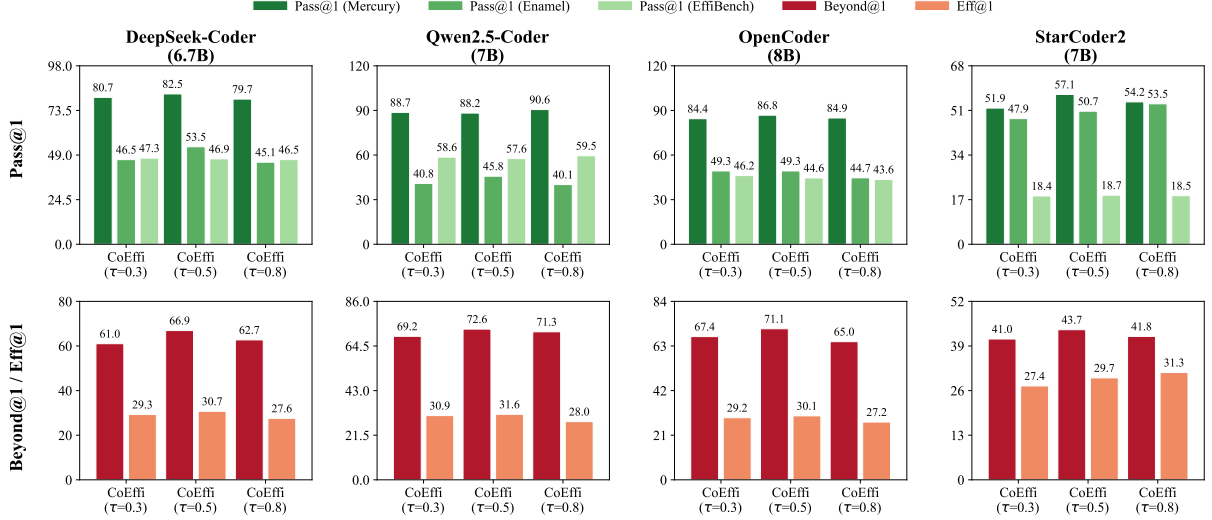
5.3 Effectiveness under Increased Difficulty

We further investigate CoEffi on medium- and high-difficulty programming scenarios. Table 3 shows part of the results.

On Mercury’s medium/hard subset, CoEffi improves Beyond@1 by 10.7%–33.8% and Pass@1 by 13.7%–29.8%. Compared to the overall performance on efficiency section, three out of four models show even larger Beyond@1 gains under increased difficulty. On EffiBench, CoEffi achieves the largest NET reductions (average improvement rising from 15.9% to 23.1%), and memory overhead is substantially compressed, especially on OpenCoder and StarCoder2. Functional correctness on EffiBench shows mixed but acceptable

Table 3: Partial comparison of efficiency and functional correctness on medium- and high-difficulty subsets.

Model	Variant	pass@1 (Mercury) ↑	Beyond@1 ↑	pass@1 (EffiBench) ↑	ET (s) ↓	NET ↓	MU (MB) ↓	NMU ↓	TMU (MB-s) ↓	NTMU ↓
DeepSeek-Coder (6.7B)	Instruct	0.6759	0.5273	0.4480	0.28	3.32	34.40	1.04	12.65	3.14
	DPO	0.6966 (↑3.1%)	0.5456 (↑3.5%)	0.4380 (↓2.2%)	0.27 (↓3.6%)	3.16 (↓4.8%)	39.36 (↑14.4%)	1.16 (↑11.5%)	22.56 (↑78.3%)	5.13 (↑63.4%)
	SFT	0.7241 (↑7.1%)	0.5772 (↑9.5%)	0.4340 (↓3.1%)	0.24 (↓14.3%)	2.77 (↓16.6%)	35.81 (↑4.1%)	1.00 (↓3.8%)	9.85 (↓22.1%)	1.92 (↓38.9%)
	SkeldPO	0.7103 (↑5.1%)	0.5818 (↑10.3%)	0.4370 (↓2.5%)	0.24 (↓14.3%)	2.74 (↓17.5%)	40.42 (↑17.5%)	1.15 (↑10.6%)	30.54 (↑141.4%)	6.14 (↑95.5%)
	CoEffi	0.7862 (↑16.3%)	0.6150 (↑16.6%)	0.4270 (↓4.7%)	0.22 (↓21.4%)	2.56 (↓22.9%)	38.00 (↑10.5%)	1.13 (↑8.7%)	19.15 (↑51.4%)	4.40 (↑40.1%)
Qwen2.5-Coder (7B)	Instruct	0.7448	0.6076	0.4900	0.21	2.47	42.84	1.29	33.84	7.92
	DPO	0.7862 (↑5.6%)	0.6363 (↑4.7%)	0.5380 (↑9.8%)	0.18 (↓14.3%)	2.08 (↓15.8%)	35.19 (↓17.9%)	1.05 (↓18.6%)	11.51 (↓66.0%)	2.62 (↓66.9%)
	SFT	0.8138 (↑9.3%)	0.6503 (↑7.0%)	0.5030 (↑2.7%)	0.18 (↓14.3%)	2.01 (↓18.6%)	36.62 (↓14.5%)	1.08 (↓16.3%)	15.00 (↓55.7%)	3.17 (↓60.0%)
	SkeldPO	0.8094 (↑8.7%)	0.6441 (↑6.0%)	0.4510 (↓8.0%)	0.17 (↓19.0%)	1.90 (↓23.1%)	40.31 (↓5.9%)	1.17 (↑9.3%)	32.63 (↓3.6%)	6.93 (↓12.5%)
	CoEffi	0.8483 (↑13.9%)	0.6725 (↑10.7%)	0.5420 (↑10.6%)	0.15 (↓28.6%)	1.84 (↓25.5%)	44.87 (↑4.7%)	1.33 (↑3.1%)	33.70 (↓0.4%)	7.62 (↓3.8%)

Figure 3: Pass@1, Beyond@1, and Eff@1 vs. initial dynamic threshold τ_0 .

trends, consistent with the previous subsection. These results confirm that CoEffi’s efficiency benefits are consistent and even amplified in more challenging scenarios. Our case study shows how rising problem difficulty manifests on the problems, and how CoEffi stays effective. Full tables and case studies are provided in Appendix E.

5.4 Component Ablation and Stability

We analyze CoEffi’s sensitivity to its initial dynamic threshold (τ_0). Figure 3 shows the effect of varying τ_0 on functional and comprehensive metrics. More results are provided in Appendix F.

The dynamic threshold governs the intervention intensity of the efficiency expert model. Relative to the initial dynamic threshold $\tau_0 = 0.5$, a loose initial dynamic threshold ($\tau_0 = 0.3$) weakens regulation, leading to moderate declines in Beyond@1 and Eff@1; a strict initial dynamic threshold ($\tau_0 = 0.8$) causes resource overhead to rebound and erodes comprehensive efficiency. The default $\tau_0 = 0.5$ offers the best balance across the tested settings.

We further evaluate the contributions of the main components on Mercury. Compared with

the best fixed-threshold control, full CoEffi improves Pass@1 by 0.5–2.8 percentage points and Beyond@1 by 1.9–4.6 percentage points across the four models. Full CoEffi also outperforms the dynamic-threshold variant with a deterministic cutoff on both metrics for all four models, demonstrating the contribution of stochastic acceptance. Compared with collaborative decoding without efficiency-expert fine-tuning, full CoEffi improves Pass@1 by 5.2–24.1 percentage points and Beyond@1 by 5.2–19.4 percentage points. The control without efficiency fine-tuning remains close to the original instruction-tuned models, with absolute differences within 2.36 percentage points for Pass@1 and 1.12 percentage points for Beyond@1. Full CoEffi also outperforms the standalone expert and SkeldPO-trained base-model controls across all four configurations.

We conduct three independent decoding runs for each model on Mercury. Across the four models, the sample standard deviation ranges from 0.0031 to 0.0170 for Pass@1 and from 0.0011 to 0.0112 for Beyond@1, showing limited variation across runs.

5.5 Qualitative Analysis

We conducted a qualitative analysis of CoEffi based on three representative cases (detailed in Appendix G). CoEffi significantly improves code generation: it suppresses syntactic hallucinations, guides efficient loop implementations, produces rare algorithmic optimizations, and achieves execution efficiency superior to both the vanilla instruction-tuned models and fine-tuning baselines. However, in one problem, CoEffi reproduced an off-by-one indexing error present in its expert model’s training data, indicating that CoEffi may introduce systematic biases from the training data source in some scenarios. Overall, CoEffi acts as a high-level algorithmic heuristic, yet the quality of training data can affect its performance in fine-grained generation scenarios.

6 Conclusion

In this paper, we introduced **CoEffi**, a dynamic collaborative decoding framework that fills the gap in efficiency optimization for code generation during the inference phase of code LLMs. CoEffi consistently improves code efficiency metrics across evaluated settings, and enhances functional correctness on Mercury and ENAMEL, while exhibiting model-dependent correctness trade-offs on EffiBench. A family specific trained expert can be reused across compatible base-model scales within the same family.

Limitations

In practical deployments, the inference latency of code Large Language Models (LLMs) is a critical metric. Since CoEffi requires synchronized inference between the base model and the expert model, it inevitably introduces additional time overhead compared to a standalone instruction-tuned model. Across all evaluated model families, the inference time of CoEffi is higher than that of the baselines, with the latency amplification generally ranging from $1.16\times$ to $1.40\times$.

The root cause of this overhead lies in the serial nature of autoregressive token generation, which is further compounded by the synchronized dual-model mechanism in CoEffi. Nevertheless, we argue that this overhead remains within an acceptable range for practical applications. On one hand, we implement a KV-Cache mechanism (Kwon et al., 2023) to store the key-value states of historical tokens, effectively avoiding redundant computations.

On the other hand, as CoEffi serves as an effective method for code performance enhancement, the substantial efficiency gains it delivers justify the moderate increase in inference latency.

Based on these observations, we recommend prioritizing the smallest variant within the same model family as the efficiency expert model during practical deployment, while dynamically adjusting the expert’s scale based on specific application scenarios. This strategy not only minimizes dependencies on data scale and compute during training, but also keeps the response time within a reasonable bound. To further mitigate this limitation, our future work will explore integrating accelerated collaborative inference techniques (Fu et al., 2025) with CoEffi to alleviate the latency bottleneck imposed by synchronized inference.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 62372071 and in part by the Fundamental Research Funds for the Central Universities under Grant No. 2022CDJDX-005.

References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Meng Cao, Mehdi Fatemi, Jackie Chi Kit Cheung, and Samira Shabanian. 2023. Systematic rectification of language models via dead-end analysis. *arXiv preprint arXiv:2302.14003*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Naveen Sachdeva, Inderjit Dhillon, Marcel Bliestein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.
- Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented code change automation: The fusion of LLMs and transformation by example. *Proceedings of the ACM on Software Engineering*, 1(FSE):631–653.

- Mingzhe Du, Anh Tuan Luu, Yue Liu, Yuhao Qing, Dong Huang, Xinyi He, Qian Liu, Zejun Ma, and See-kiong Ng. 2026. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. *Advances in Neural Information Processing Systems*, 38:8979–9011.
- Mingzhe Du, Luu A Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37:16601–16622.
- Jiale Fu, Yuchu Jiang, Junkai Chen, Jiaming Fan, Xin Geng, and Xu Yang. 2025. Fast large language model collaborative decoding via speculation. *arXiv preprint arXiv:2502.01662*.
- Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, and 1 others. 2024. Deepseek-coder: when the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.
- Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. 2024a. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems*, 37:84482–84522.
- Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. 2024b. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544.
- Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. 2024c. Efficoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. *arXiv preprint arXiv:2410.10209*.
- Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, and 1 others. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.
- Juyong Jiang, Jiasi Shen, Sunghun Kim, Kang Min Yoo, Jeonghoon Kim, and Sungju Kim. 2026. Reflexi-coder: Teaching large language models to self-reflect on generated code and self-correct it via reinforcement learning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 37543–37562.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Seonghyeon Lee, Heejae Chon, Joonwon Jang, Dongha Lee, and Hwanjo Yu. 2025. How diversely can language models solve problems? exploring the algorithmic diversity of model-generated code. In *EMNLP (Findings)*, pages 152–167.
- Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A Smith, and Yejin Choi. 2021. Dexperts: Decoding-time controlled text generation with experts and anti-experts. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6691–6706.
- 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. *Advances in neural information processing systems*, 36:21558–21572.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.
- Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas Reps, Swarat Chaudhuri, and Christopher Jermaine. 2021. Neural program generation modulo static analysis. *Advances in Neural Information Processing Systems*, 34:18984–18996.

- Ruizhong Qiu, Weiliang Zeng, James Ezick, Christopher Lott, and Hanghang Tong. 2025. How efficient is llm-generated code? a rigorous & high-standard benchmark. In *International Conference on Learning Representations*, volume 2025, pages 2233–2261.
- Zejiang Shen, Hunter Lang, Bailin Wang, Yoon Kim, and David Sontag. 2024. Learning to decode collaboratively with multiple language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12974–12990.
- Stack Overflow. 2025. [AI | 2025 Stack Overflow Developer Survey](#). Online. Accessed: 2026-04-18.
- Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8984–8991.
- Tong Ye, Weigang Huang, Xuhong Zhang, Tengfei Ma, Peiyu Liu, Jianwei Yin, and Wenhai Wang. 2025. Llm4effi: Leveraging large language models to enhance code efficiency and correctness. *arXiv preprint arXiv:2502.18489*.
- Yu Yu and Chen Lyu. 2026. Skeldpo: A skeleton-guided direct preference optimization framework for efficient code generation. In *Proceedings of the 2026 34th IEEE/ACM International Conference on Program Comprehension*, pages 367–378.
- Zhuohao Yu, Weizheng Gu, Yidong Wang, Xingru Jiang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. 2024. Reasoning through execution: Unifying process and outcome rewards for code generation. *arXiv preprint arXiv:2412.15118*.
- Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2025a. 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)*, pages 15854–15871.
- Xiangyu Zhang, Yu Zhou, Guang Yang, Wei Cheng, and Taolue Chen. 2025b. Beyond sequences: Two-dimensional representation and dependency encoding for code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6157–6172.
- Tianqi Zhong, Quan Wang, Jingxuan Han, Yongdong Zhang, and Zhendong Mao. 2023. Air-decoding: Attribute distribution reconstruction for decoding-time controllable text generation. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 8233–8248.

A Detailed Definition of the Loss Formula

This appendix restates the original formulation of SkelDPO (Yu and Lyu, 2026), including the standard DPO loss, the code-level and skeleton-level preference losses, and the joint training objective.

A.1 Standard DPO Loss

Direct Preference Optimization (DPO) optimizes over preference pairs (y^+, y^-) for a given input x via a contrastive objective:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x, y^+, y^-)} \left[\log \sigma \left(\beta \left(\begin{aligned} &\log \pi_{\theta}(y^+ | x) \\ &- \log \pi_{\theta}(y^- | x) \\ &- \log \pi_{\text{ref}}(y^+ | x) \\ &+ \log \pi_{\text{ref}}(y^- | x) \end{aligned} \right) \right) \right], \quad (8)$$

where π_{θ} is the current model, π_{ref} is a fixed reference model, $\beta > 0$ is a temperature parameter, and $\sigma(\cdot)$ is the sigmoid function.

A.2 Code-level Preference Loss $\mathcal{L}_{\text{code}}$

Applied to full, executable program pairs (y^+, y^-) that differ in efficiency while holding correctness comparable:

$$\mathcal{L}_{\text{code}}(\theta) = \mathcal{L}_{\text{DPO}}(\theta; (x, y^+, y^-)). \quad (9)$$

A.3 Skeleton-level Preference Loss $\mathcal{L}_{\text{skeleton}}$

Applied to skeleton pairs (S^+, S^-) extracted from the corresponding code versions:

$$\mathcal{L}_{\text{skeleton}}(\theta) = \mathcal{L}_{\text{DPO}}(\theta; (x, S^+, S^-)). \quad (10)$$

A.4 Joint Objective of SkelDPO

For each problem, SkelDPO combines the two losses with a convex coefficient $\alpha \in [0, 1]$:

$$\mathcal{L}_{\text{SkelDPO}}(\theta) = \alpha \cdot \mathcal{L}_{\text{code}}(\theta) + (1 - \alpha) \cdot \mathcal{L}_{\text{skeleton}}(\theta). \quad (11)$$

This joint objective encourages the model to generate code that is both more correct and more efficient at the whole-program level.

B Hyperparameter Details

Table 4 lists all key hyperparameters used in our experiments.

C Detailed Experimental Setup

This appendix provides the complete details of our experimental setup, including dataset construction, benchmark specifications, contamination control, training configurations, collaborative inference settings, and formal metric definitions.

Table 4: Hyperparameters for CoEffi and baselines.

Component	Parameter	Value
CoEffi (inference)	Initial dynamic threshold τ_0	0.5 (default)
	Base temperature T_{base}	0.6
	Expert temperature T_{expert}	0.8
	Sliding window size	10
	Adjustment step	0.10
Fine-tuning (DPO/SFT/SkelDPO)	Learning rate	5×10^{-6}
	Batch size	8
	Epochs	1.5
	Warmup ratio	0.1
Evaluation	Max new tokens	512 (EffiBench), 1024 (others)
	Number of samples (Pass@1)	1
	GPU	1× NVIDIA RTX PRO 6000 (96GB)

C.1 Dataset Construction Details

Mercury (Du et al., 2024). From 1,889 LeetCode problems, 1,633 tasks with ≥ 2 accepted solutions (Python built-ins plus `TreeNode`, `ListNode`, unique outputs) are retained. Top-5 runtime-discrepant solution pairs form DPO training pairs (faster preferred). 256 difficulty-balanced tasks are held out as Mercury-eval.

EffiCoder (Huang et al., 2024c). Its Effi-Instruct split aggregates nine HuggingFace datasets. After SelfCodeAlign filtering, 33,489 Python tasks remain; 9,451 are selected for SFT. For each task, multiple LLMs generate candidates; the solution with lowest execution time and memory is chosen. Missing tests are built via GPT-3.5-turbo.

SkelDPO (Yu and Lyu, 2026). Built from the APPS dataset, candidate solutions are executed per problem to check correctness and measure runtime. The top-5 fastest correct solutions are labeled efficient, the slowest 5 inefficient. Code skeletons are then extracted by computing token-level intersections across efficient (and inefficient) implementations, replacing non-intersecting tokens with placeholders. After filtering, the resulting dataset comprises 2,880 training problems, each with an efficient/inefficient code pair and a corresponding skeleton pair.

C.2 Benchmark Details

Mercury-eval. Comprises 256 difficulty-balanced tasks (88 Easy, 81 Medium, 87 Hard); runtimes are compared with accepted reference distributions.

ENAMEL (Qiu et al., 2025). Provides a stricter HumanEval-style (Chen et al., 2021) framework of 142 problems (from HumanEval/HumanEval+ excluding $\Theta(1)$ complexity), with expert-written efficient references and strong test generators under calibrated input sizes.

EffiBench (Huang et al., 2024b). Contains 1,000 efficiency-critical LeetCode problems (171 Easy, 589 Medium, 240 Hard), measuring execution time and memory relative to canonical solutions.

Increased Difficulty (§5.3). We remove Easy instances: from Mercury-eval we retain Medium and Hard tasks with non-empty static tests; from EffiBench we keep Medium and Hard problems. ENAMEL is excluded due to missing three-tier difficulty annotations.

C.3 Data Overlap and Contamination Control

We use the officially released benchmark versions without modification. To mitigate data contamination, we adopt the following measures: (1) Mercury-eval is disjoint from the Mercury training set by construction; (2) EffiCoder authors confirm no exact duplicates between their training data and EffiBench, with negligible vocabulary overlap ($\ll 0.2\%$); (3) SkelDPO is trained on the APPs dataset, whose problem sources differ from those of LeetCode/HumanEval-based benchmarks, reducing the risk of direct overlap; (4) We confirm that ENAMEL and EffiBench have no known overlap with our training sets.

C.4 Training Configuration

SFT, DPO, and SkelDPO use LoRA (rank 16, alpha 32, dropout 0.05) on attention and feed-forward layers, learning rate $5e-6$, max sequence length 2048. DPO and SFT train 1.5 epochs (DPO $\beta = 0.1$); SkelDPO follows its original settings. All efficiency experts are uniformly trained with SkelDPO: its dense efficiency-critical skeletons concentrate token preferences on efficiency hotspots, giving CoEffi’s lightweight intervention high-value gains per acceptance.

C.5 Collaborative Inference Configuration

The expert is attached via LoRA, both float16. The base model generates with nucleus sampling ($p = 0.95$) at temperature 0.6, max new tokens per benchmark default. The expert operates at temperature 0.8. The initial dynamic threshold τ_0 is set to 0.5, bounded within $[0.3, 1.2]$, with window $w = 10$ and step 0.10.

C.6 Evaluation Metrics and Formal Definitions

Functional Correctness: Pass@1. Pass@1 is the fraction of tasks passing all tests. Let $g_i \in$

$\{0, 1\}$ be the correctness indicator for problem i :

$$\text{Pass@1} = \frac{1}{N} \sum_{i=1}^N g_i. \quad (12)$$

Mercury: Beyond@1. For task n , with reference runtimes R^n and generated runtime r^n :

$$p^n = \frac{\max(R^n) - \text{clip}(r^n, \min(R^n), \max(R^n))}{\max(R^n) - \min(R^n)}, \quad (13)$$

$$\text{Beyond@1} = \frac{1}{N} \sum_{n=1}^N p^n. \quad (14)$$

Failed solutions receive $r^n = +\infty$ ($p^n = 0$).

ENAMEL: Eff@1. The average efficiency score against the expert reference:

$$\text{Eff@1} = \frac{1}{N} \sum_{i=1}^N e_i. \quad (15)$$

EffiBench Resource Metrics. We report Execution Time (ET), Normalized ET (NET), Max Memory Usage (MU), Normalized MU (NMU), Total Memory Usage (TMU), and Normalized TMU (NTMU). NET, NMU, and NTMU are ratios to the canonical solution; lower is better for all six metrics.

Execution Time (ET). Average execution time:

$$\text{ET} = \frac{1}{N} \sum_{i=1}^N T_{\text{code}}(i), \quad (16)$$

where $T_{\text{code}}(i)$ is the execution time of the generated code for problem i , and N is the total number of evaluated problems.

Normalized Execution Time (NET). Normalized execution time:

$$\text{NET} = \frac{1}{N} \sum_{i=1}^N \frac{T_{\text{code}}(i)}{T_{\text{canonical}}(i)}, \quad (17)$$

where $T_{\text{canonical}}(i)$ is the execution time of the canonical (reference) solution. $\text{NET} > 1$ indicates the generated code is slower than the canonical solution.

Max Memory Usage (MU). Average peak memory usage:

$$\text{MU} = \frac{1}{N} \sum_{i=1}^N M_{\text{code}}(i), \quad (18)$$

where $M_{\text{code}}(i)$ is the peak memory usage of the generated code across all test cases for problem i .

Normalized Max Memory Usage (NMU). Normalized peak memory usage:

$$\text{NMU} = \frac{1}{N} \sum_{i=1}^N \frac{M_{\text{code}}(i)}{M_{\text{canonical}}(i)}, \quad (19)$$

where $M_{\text{canonical}}(i)$ is the peak memory usage of the canonical solution.

Total Memory Usage (TMU). Total memory usage, defined as the area under the memory-time curve:

$$\text{TMU} = \frac{1}{N} \sum_{i=1}^N \int_0^{T_{\text{total}}(i)} M_i(t) dt, \quad (20)$$

where $M_i(t)$ is the memory usage over time for problem i , and $T_{\text{total}}(i)$ is the total execution time. The integral is computed numerically (e.g., via the trapezoidal rule).

Normalized Total Memory Usage (NTMU). Normalized total memory usage:

$$\text{NTMU} = \frac{1}{N} \sum_{i=1}^N \frac{\text{TMU}_{\text{code}}(i)}{\text{TMU}_{\text{canonical}}(i)}, \quad (21)$$

where $\text{TMU}_{\text{code}}(i)$ and $\text{TMU}_{\text{canonical}}(i)$ are the total memory usages of the generated code and the canonical solution for problem i , respectively.

D Detailed Data for the Analysis of CoEffi’s Correctness Improvement

Table 5: Comparison of SkelDPO across scales and CoEffi on Mercury.

Model	Variant	Pass@1 ↑	Beyond@1 ↑
DeepSeek-Coder	SkelDPO (1.3B)	0.5708	0.4278
	SkelDPO (6.7B)	0.8019	0.6292
	CoEffi (6.7B)	0.8255	0.6692
Qwen2.5-Coder	SkelDPO (1.5B)	0.5755	0.4635
	SkelDPO (7B)	0.8396	0.6842
	CoEffi (7B)	0.8821	0.7263
OpenCoder	SkelDPO (1.5B)	0.7123	0.5575
	SkelDPO (8B)	0.8538	0.6882
	CoEffi (8B)	0.8679	0.7110
StarCoder2	SkelDPO (3B)	0.2925	0.2011
	SkelDPO (7B)	0.5047	0.3822
	CoEffi (7B)	0.5708	0.4374

1. license-key-formatting

Error: Baseline uses list insert in a loop causing $O(N^2)$ time.

Fix: CoEffi uses string slicing with join to rebuild the string in linear time.

Attribution: Expert injection.

2. longest-palindrome

Error: Baseline parity check fails with multiple odd-frequency characters.

Fix: CoEffi adds all even counts fully, adds count minus one for odds, and adds one if any odd exists.

Attribution: Expert injection.

3. cut-off-trees-for-golf-event

Error: Baseline uses Dijkstra for each step in a large matrix.

Fix: CoEffi switches to standard BFS for shortest paths on unweighted graphs.

Attribution: Butterfly effect: baseline wrote the bfs skeleton, expert patched loop-state updates via «E».

4. count-of-range-sum

Error: Baseline binary search direction is reversed.

Fix: CoEffi corrects the inequality.

Attribution: Expert injection.

5. preimage-size-of-factorial-zeroes-function

Error: Baseline has redundant dual while loops with complex boundary conditions.

Fix: CoEffi abstracts a find_k helper and returns the difference between k+1 and k.

Attribution: Expert injection.

6. palindrome-pairs

Error: Baseline uses a memory-intensive custom hand-written trie.

Fix: CoEffi replaces it with a hash table storing reversed strings and matches via prefix and suffix splitting.

Attribution: Expert injection.

7. word-break-ii

Error: Baseline uses backward DP with heavy string strip() concatenation risking memory limit.

Fix: CoEffi uses forward DP with list storage and final join output.

Attribution: Expert injection.

8. power-of-three

Error: Baseline uses math.log10 causing float precision errors on large inputs.

Fix: CoEffi provides a loop divide-by-three method or a constant-time magic number check `1162261467 % n == 0`.

Attribution: Expert injection.

9. sliding-window-median

Error: Baseline uses SortedList.remove() which is costly for large windows.

Fix: CoEffi adopts a dual-heap architecture with lazy deletion and heap balancing.

Attribution: Expert injection.

10. 24-game

Error: Baseline has poorly structured nested loops with pop() missing permutations.

Fix: CoEffi passes a newly constructed list with the operation result to DFS.

Attribution: Expert injection.

11. integer-to-english-words

Error: Baseline relies on a large dictionary with reverse modulo failing on zero and carry.

Fix: CoEffi splits logic into hierarchical helper functions for digit processing.

Attribution: Expert injection.

12. k-inverse-pairs-array

Error: Baseline DP subtraction lacks negative modulo handling.

Fix: CoEffi adds MOD before taking modulo to avoid negative values.

Attribution: Expert injection.

13. course-schedule-iii

Error: Baseline greedy replacement condition is too rigid.

Fix: CoEffi simplifies by pushing all to a max-heap and

popping the largest when total exceeds deadline.

Attribution: Expert injection.

14. strange-printer

Error: Baseline interval DP update is overly complex with fragile boundaries.

Fix: CoEffi moves the equality check early to simplify the state transition.

Attribution: Expert injection.

15. word-ladder-ii

Error: Baseline double-loop string concatenation causes time limit exceeded.

Fix: CoEffi uses `defaultdict(list)` to maintain a layer graph of old and new states.

Attribution: Expert injection.

16. sliding-puzzle

Error: Baseline dynamically computes 2D coordinate transforms in every BFS step.

Fix: CoEffi hardcodes a one-dimensional moves dictionary for all six positions.

Attribution: Expert injection.

17. strong-password-checker

Error: Baseline has conflicting delete and replace logic in a lengthy else branch.

Fix: CoEffi categorizes length greater than twenty and correctly computes the offset between replace and delete.

Attribution: Expert injection.

18. zigzag-conversion

Error: Baseline uses relative state flipping which is prone to edge-case failures.

Fix: CoEffi uses absolute state judgment based on current row position.

Attribution: Expert injection.

E Effectiveness under Increased Difficulty: Detailed Data and Case Analyses

E.1 Complete Metrics on Medium-and-Hard Subsets

Table 6 reports all efficiency and correctness metrics for the medium-and-hard subsets of Mercury and EffiBench, corresponding to the summary in the main paper.

E.2 Case Studies

We select two problems from the Mercury benchmark, “Plus One” (easy) and “Russian Doll Envelopes” (hard), and conduct a comparative analysis between the vanilla instruction-tuned models and CoEffi implementations on DeepSeek-Coder. As shown in Figure 4 and Figure 5, for Plus One, the vanilla instruction-tuned models adopts an integer conversion approach ($O(n)$ time, $O(n)$ space, with large constant overhead), whereas CoEffi uses in-place carry simulation ($O(n)$ time, $O(1)$ space, avoiding type conversions). Both share the same asymptotic complexity, but the constant-factor optimization causes `beyond_p` to jump from 0.224 to 1.000, indicating that CoEffi can generate solutions with superior constant factors on simple

problems where asymptotic improvement is hard to achieve. For Russian Doll Envelopes, the vanilla instruction-tuned models uses $O(n^2)$ dynamic programming and suffers from severe timeout, while CoEffi employs patience sorting with sorting and binary search, reducing the complexity to $O(n \log n)$ and causing `beyond_p` to leap from 0.000 to 0.984, demonstrating that CoEffi can generate solutions with superior asymptotic complexity on harder problems.

F Parameter Sensitivity Analysis: Detailed Data Bar Chart

Figure 6 shows the effect of varying τ_0 on EffiBench efficiency metrics. The default $\tau_0 = 0.5$ provides the best trade-off.

G Qualitative Analysis

Corresponding Cases: base-7, word-search-ii, first-missing-positive

G.1 Case 1: base-7 (Base-7 Conversion)

As shown in Figure 7. In the base-7 conversion task, the base model produced severe syntactic hallucinations (e.g., generating the non-executable `str(int(str), 10, 7)`); DPO mistakenly used the octal conversion function `oct()`; while SkelDPO simply removed the minus sign and converted to decimal, completely ignoring the core conversion requirement. SFT generated logically correct code but used `while num > 0:` in the loop condition, which introduces additional comparison overhead over many iterations. In contrast, CoEffi successfully generated the standard iterative modulo algorithm and simplified the loop condition to `while num:` (leveraging Python’s truth value testing mechanism), thereby reducing unnecessary comparisons. Consequently, CoEffi achieved a Beyond score of 0.8718, significantly outperforming SFT’s 0.4182. This result demonstrates that CoEffi not only effectively suppresses syntactic hallucinations but also guides the model toward more efficient low-level implementation patterns.

G.2 Case 2: word-search-ii (Word Search II)

As shown in Figure 8. In the word search II task, the base model, SFT, DPO, and SkelDPO all implemented basic Trie matching or simple pruning, but none performed physical node deletion. DPO only marked words as found (`del node['#']`), while SFT had no pruning at

Table 6: Full results on medium- and high-difficulty subsets.

Model	Variant	Pass@1 ↑ (Mercury)	Beyond@1 ↑	Pass@1 ↑ (EffiBench)	ET (s) ↓	NET ↓	MU (MB) ↓	NMU ↓	TMU (MB-s) ↓	NTMU ↓
DeepSeek-Coder (6.7B)	Instruct	0.6759	0.5273	0.4480	0.28	3.32	34.40	1.04	12.65	3.14
	DPO	0.6966 (↑3.1%)	0.5456 (↑3.5%)	0.4380 (↓2.2%)	0.27 (↓3.6%)	3.16 (↓4.8%)	39.36 (↑14.4%)	1.16 (↑11.5%)	22.56 (↑78.3%)	5.13 (↑63.4%)
	SFT	0.7241 (↑7.1%)	0.5772 (↑9.5%)	0.4340 (↓3.1%)	0.24 (↓14.3%)	2.77 (↓16.6%)	35.81 (↑4.1%)	1.00 (↓3.8%)	9.85 (↓22.1%)	1.92 (↓38.9%)
	SkelDPO	0.7103 (↑5.1%)	0.5818 (↑10.3%)	0.4370 (↓2.5%)	0.24 (↓14.3%)	2.74 (↓17.5%)	40.42 (↑17.5%)	1.15 (↑10.6%)	30.54 (↑141.4%)	6.14 (↑95.5%)
	CoEffi	0.7862 (↑16.3%)	0.6150 (↑16.6%)	0.4270 (↓4.7%)	0.22 (↓21.4%)	2.56 (↓22.9%)	38.00 (↑10.5%)	1.13 (↑8.7%)	19.15 (↑51.4%)	4.40 (↑40.1%)
Qwen2.5-Coder (7B)	Instruct	0.7448	0.6076	0.4900	0.21	2.47	42.84	1.29	33.84	7.92
	DPO	0.7862 (↑5.6%)	0.6363 (↑4.7%)	0.5380 (↑9.8%)	0.18 (↓14.3%)	2.08 (↓15.8%)	35.19 (↓17.9%)	1.05 (↓18.6%)	11.51 (↓66.0%)	2.62 (↓66.9%)
	SFT	0.8138 (↑9.3%)	0.6503 (↑7.0%)	0.5030 (↓2.7%)	0.18 (↓14.3%)	2.01 (↓18.6%)	36.62 (↓14.5%)	1.08 (↓16.3%)	15.00 (↓55.7%)	3.17 (↓60.0%)
	SkelDPO	0.8094 (↑8.7%)	0.6441 (↑6.0%)	0.4510 (↓8.0%)	0.17 (↓19.0%)	1.90 (↓23.1%)	40.31 (↓5.9%)	1.17 (↑9.3%)	32.63 (↓3.6%)	6.93 (↓12.5%)
	CoEffi	0.8483 (↑13.9%)	0.6725 (↑10.7%)	0.5420 (↑10.6%)	0.15 (↓28.6%)	1.84 (↓25.5%)	44.87 (↑4.7%)	1.33 (↑3.1%)	33.70 (↓0.4%)	7.62 (↓3.8%)
OpenCoder (8B)	Instruct	0.6483	0.5178	0.4210	0.22	2.26	82.39	2.34	141.19	26.70
	DPO	0.7793 (↑20.2%)	0.6387 (↑23.3%)	0.4190 (↓0.5%)	0.18 (↓18.2%)	2.04 (↓9.7%)	41.12 (↓50.1%)	1.24 (↓47.0%)	27.12 (↓80.8%)	6.04 (↓77.4%)
	SFT	0.7586 (↑17.0%)	0.6005 (↑16.0%)	0.3140 (↓25.4%)	0.17 (↓22.7%)	1.89 (↓16.4%)	38.86 (↓52.8%)	1.07 (↓54.3%)	14.63 (↓89.6%)	2.70 (↓89.9%)
	SkelDPO	0.7448 (↑14.9%)	0.5982 (↑15.5%)	0.4340 (↑3.1%)	0.19 (↓13.6%)	2.09 (↓7.5%)	33.04 (↓59.9%)	1.04 (↓55.6%)	11.57 (↓91.8%)	3.16 (↓88.2%)
	CoEffi	0.8414 (↑29.8%)	0.6927 (↑33.8%)	0.4250 (↑1.0%)	0.15 (↓31.8%)	1.74 (↓23.0%)	32.81 (↓60.2%)	1.00 (↓57.3%)	6.24 (↓95.6%)	1.53 (↓94.3%)
StarCoder2 (7B)	Instruct	0.3517	0.2642	0.1240	0.77	7.10	351.05	7.23	947.43	87.04
	DPO	0.3793 (↑7.8%)	0.2890 (↑9.4%)	0.1340 (↑8.1%)	0.69 (↓10.4%)	6.97 (↓1.8%)	202.96 (↓42.2%)	5.33 (↓26.3%)	539.33 (↓43.1%)	77.87 (↓10.5%)
	SFT	0.3655 (↑3.9%)	0.2803 (↑6.1%)	0.1310 (↑5.6%)	0.69 (↓10.4%)	5.92 (↓16.6%)	73.08 (↓79.2%)	1.39 (↓80.8%)	80.72 (↓91.5%)	6.48 (↓92.6%)
	SkelDPO	0.3931 (↑11.8%)	0.2980 (↑12.8%)	0.1470 (↑18.5%)	0.71 (↓7.8%)	7.00 (↓1.4%)	180.36 (↓48.6%)	5.00 (↓30.8%)	399.32 (↓57.9%)	83.02 (↓4.6%)
	CoEffi	0.4000 (↑13.7%)	0.3104 (↑17.5%)	0.1150 (↓7.3%)	0.57 (↓26.0%)	5.62 (↓20.8%)	135.60 (↓61.4%)	3.07 (↓57.5%)	322.62 (↓65.9%)	35.21 (↓59.5%)

PROMPT	Instruct-baseline	CoEffi
<p>You are given a large integer represented as an integer array <code>digits</code>, where each <code>digits[i]</code> is the <i>i</i>th digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's. Increment the large integer by one and return the resulting array of digits.</p> <pre> class Solution(object): def plusOne(self, digits): """ :type digits: List[int] :rtype: List[int] """ # Write your code here </pre>	<pre> class Solution(object): def plusOne(self, digits): num = int(''.join(map(str, digits))) num += 1 return list(map(int, str(num))) </pre>	<pre> class Solution(object): def plusOne(self, digits): for i in range(len(digits) - 1, -1, -1): if digits[i] == 9: digits[i] = 0 else: digits[i] += 1 return digits return [1] + digits </pre>

Figure 4: Example problem from Mercury. Solutions of the vanilla instruction-tuned models and CoEffi on “Plus One” (easy difficulty) using DeepSeek-Coder 6.7B.

PROMPT	Instruct-baseline	CoEffi
<p>You are given a 2D array of integers <code>envelopes</code> where <code>envelopes[i] = [wi,hi]</code> represents the width and the height of an envelope. One envelope can fit into another if and only if both the width and height of one envelope are greater than the other envelope's width and height. Return the maximum number of envelopes you can Russian doll (i.e., put one inside the other).</p> <pre> class Solution(object): def maxEnvelopes(self, envelopes): """ :type envelopes: List[List[int]] :rtype: int """ # Write your code here </pre>	<pre> class Solution(object): def maxEnvelopes(self, envelopes): if not envelopes: return 0 envelopes.sort(key=lambda x: (x[0], -x[1])) dp = [1] * len(envelopes) for i in range(len(envelopes)): for j in range(i): if envelopes[j][1] < envelopes[i][1]: dp[i] = max(dp[i], dp[j] + 1) return max(dp) </pre>	<pre> import bisect class Solution(object): def maxEnvelopes(self, envelopes): envelopes.sort(key=lambda x: (x[0], -x[1])) dp = [] for _, height in envelopes: idx = bisect.bisect_left(dp, height) if idx == len(dp): dp.append(height) else: dp[idx] = height return len(dp) </pre>

Figure 5: Example problem from Mercury. Solutions of the vanilla instruction-tuned models and CoEffi on “Russian Doll Envelopes” (hard difficulty) using DeepSeek-Coder 6.7B.

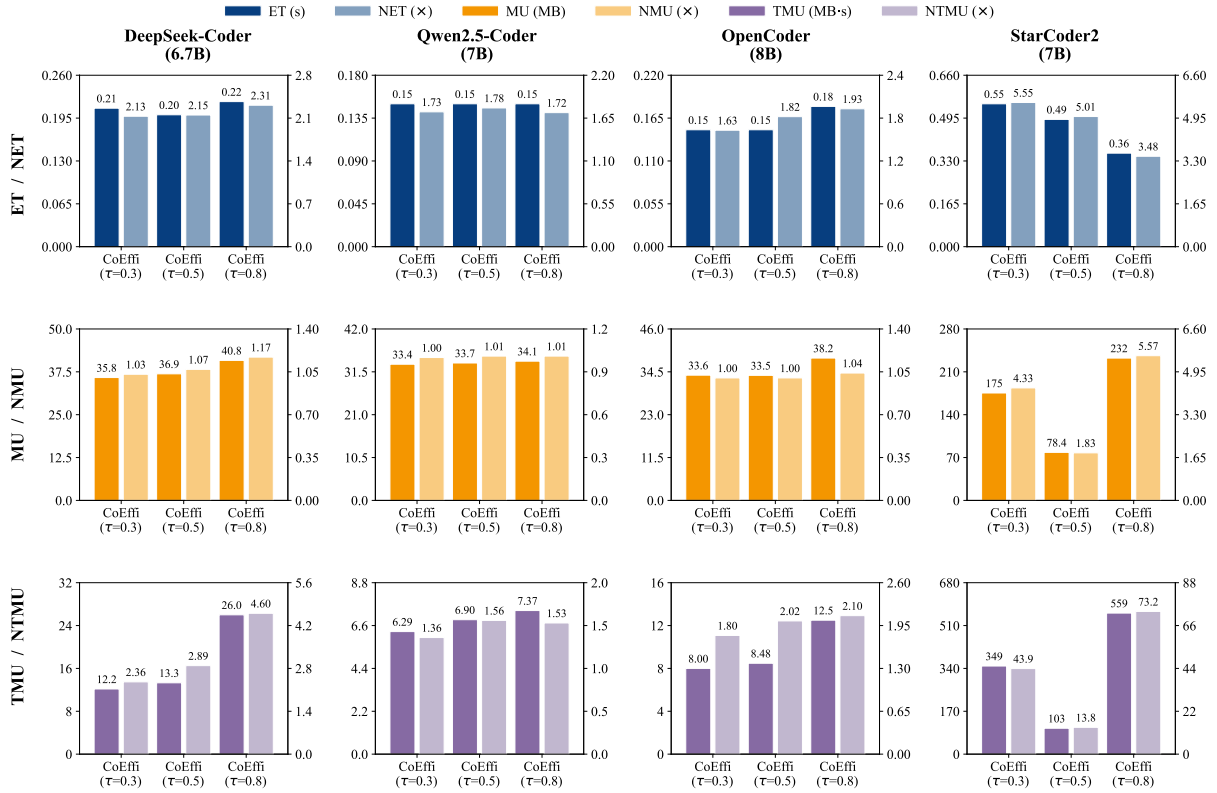


Figure 6: EffiBench efficiency metrics vs. initial dynamic threshold τ_0 .

all. CoEffi outperformed all baselines in this task, with its core advantage stemming from generating an advanced algorithmic optimization technique: dynamically deleting Trie nodes during DFS (if not currentNode: trie.pop(letter)). By promptly removing exhausted paths, this method achieves dynamic compression of the search space, significantly reducing runtime when processing large dictionaries. This technique rarely appears in training data, yet CoEffi successfully induced the model to adopt it, demonstrating its ability to select optimal complexity algorithms.

G.3 Case 3: first-missing-positive (First Missing Positive)

As shown in Figure 9. In the first missing positive task, CoEffi exhibited some fragility. It attempted the in-place hashing algorithm but made a classic off-by-one error in index mapping (using `nums[a]` instead of `nums[a-1]`), leading to logic failure. Notably, SkelDPO exhibited exactly the same error. In contrast, the base model and SFT robustly employed the “cyclic swap” algorithm, while DPO (also using the marking method) handled index mapping correctly.

In our experiments, the expert model in CoEffi was trained on SkelDPO data. Thus, the inherent index offset bias in SkelDPO was introduced into CoEffi through the collaborative decoding process. The expert model tends to guide the base model toward patterns familiar from its training data, and in this case those patterns unfortunately include the index offset error. This phenomenon suggests that CoEffi may, under specific circumstances, inherit systematic errors from its training data source.

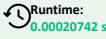
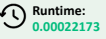
CoEffi	Instruct	DPO	SFT	SkeIDPO
<pre> class Solution(object): def convertToBase7(self,num): if num == 0: return '0' negative = num < 0 num = abs(num) result = '' while num: result = str(num % 7) + result num //= 7 return '-' + result if negative else result </pre>	<pre> class Solution(object): def convertToBase7(self, num): if num == 0: return '0' else: return str(int(str(abs(num))), 10, 7) </pre>	<pre> class Solution(object): def convertToBase7(self, num): if num == 0: return '0' else: return str(oct(abs(num))).replace ('0o', '') if num >= 0 else '-' + str(oct(abs(num))).replace ('0o', '') </pre>	<pre> class Solution(object): def convertToBase7(self, num): if num == 0: return '0' result = '' is_negative = num < 0 num = abs(num) while num > 0: result = str(num % 7) + result num //= 7 return '-' + result if is_negative else result </pre>	<pre> class Solution(object): def convertToBase7(self, num): if num == 0: return "0" else: return str(int(str(abs(num))).repl ace("-", "", 10)) if num > 0 else "-" + str(int(str(abs(num))).repl ace("-", "", 10)) </pre>
 Status: passed  Runtime: 0.00020742 s	 Status: fail  Runtime: Fail	 Status: fail  Runtime: Fail	 Status: passed  Runtime: 0.00022173 s	 Status: fail  Runtime: Fail

Figure 7: Qualitative analysis for Case 1 (base-7).

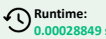
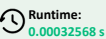
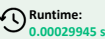
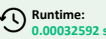
CoEffi	Instruct	DPO	SFT	SkeIDPO
<pre> class Solution(object): def findWords(self, board, words): trie = {} for word in words: node = trie for letter in word: node = node.setdefault(letter, {}) node['#'] = word self.res = [] self.board = board for i in range(len(board)): for j in range(len(board[0])): if board[i][j] in trie: self.dfs(i, j, trie) return self.res def dfs(self, i, j, trie): letter = self.board[i][j] currentNode = trie[letter] matchWord = currentNode.get('#') if matchWord: self.res.append(matchWo rd) currentNode.pop('#') self.board[i][j] = '\$' row, col = [0, 1, 0, -1], [1, 0, -1, 0] for k in range(4): x, y = i + row[k], j + col[k] if 0 <= x < len(self.board) and 0 <= y < len(self.board[0]) and self.board[x][y] in currentNode: self.dfs(x, y, currentNode) self.board[i][j] = letter if not currentNode: trie.pop(letter) </pre>	<pre> class Solution(object): def findWords(self, board, words): trie = {} for word in words: node = trie for letter in word: node = node.setdefault(letter, {}) node['#'] = word self.res = [] self.board = board for i in range(len(board)): for j in range(len(board[0])): if board[i][j] in trie: self.dfs(i, j, trie) return self.res def dfs(self, i, j, trie): letter = self.board[i][j] currentNode = trie[letter] matchWord = currentNode.pop('#', False) if matchWord: self.res.append(matchWo rd) self.board[i][j] = '\$' for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]: x, y = i + dx, j + dy if 0 <= x < len(self.board) and 0 <= y < len(self.board[0]) and self.board[x][y] in currentNode: self.dfs(x, y, currentNode) self.board[i][j] = letter if not currentNode: trie.pop(letter) </pre>	<pre> class Solution(object): def findWords(self, board, words): trie = {} for word in words: node = trie for letter in word: if letter not in node: node[letter] = {} node = node[letter] node['#'] = word def dfs(x, y, node): if '#' in node: res.append(node['#']) del node['#'] letter = board[x][y] node.next = node.get(letter) if node.next: board[x][y] = '.' for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]: nx, ny = x + dx, y + dy if 0 <= nx < len(board) and 0 <= ny < len(board[0]): dfs(nx, ny, node.next) board[x][y] = letter res = [] for x in range(len(board)): for y in range(len(board[0])): if board[x][y] in trie: dfs(x, y, trie) return res </pre>	<pre> class Solution(object): def findWords(self, board, words): trie = {} for word in words: node = trie for letter in word: node = node.setdefault(letter, {}) node['#'] = word result = [] for i in range(len(board)): for j in range(len(board[0])): self.dfs(board, i, j, trie, result) return result def dfs(self, board, i, j, trie, result): if '#' in trie: result.append(trie['#']) del trie['#'] if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or board[i][j] not in trie: return letter = board[i][j] board[i][j] = '\$' self.dfs(board, i + 1, j, trie[letter], result) self.dfs(board, i - 1, j, trie[letter], result) self.dfs(board, i, j + 1, trie[letter], result) self.dfs(board, i, j - 1, trie[letter], result) board[i][j] = letter </pre>	<pre> class Solution(object): def findWords(self, board, words): trie = {} for word in words: node = trie for letter in word: if letter not in node: node[letter] = {} node = node[letter] node['#'] = word self.res = [] self.board = board for i in range(len(board)): for j in range(len(board[0])): if board[i][j] in trie: self.dfs(i, j, trie) return self.res def dfs(self, i, j, trie): letter = self.board[i][j] currentNode = trie[letter] matchWord = currentNode.pop('#', False) if matchWord: self.res.append(matchWo rd) self.board[i][j] = '\$' for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]: x, y = i + dx, j + dy if 0 <= x < len(self.board) and 0 <= y < len(self.board[0]) and self.board[x][y] in currentNode: self.dfs(x, y, currentNode) self.board[i][j] = letter if not currentNode: trie.pop(letter) </pre>
 Status: passed  Runtime: 0.00028849 s	 Status: passed  Runtime: 0.00032568 s	 Status: passed  Runtime: 0.00029945 s	 Status: passed  Runtime: 0.00032592 s	 Status: passed  Runtime: 0.00032306 s

Figure 8: Qualitative analysis for Case 2 (word-search-ii).

 CoEffi	 Instruct	 DPO	 SFT	 SkeIDPO
<pre> class Solution(object): def firstMissingPositive(self, nums): if not nums: return 1 n = len(nums) for i in range(n): if nums[i] <= 0 or nums[i] > n: nums[i] = 1 for i in range(n): a = abs(nums[i]) if a == n: nums[0] = -abs(nums[0]) else: nums[a] = -abs(nums[a]) for i in range(1, n): if nums[i] > 0: return i if nums[0] > 0: return n + 1 return 1 </pre>	<pre> class Solution(object): def firstMissingPositive(self, nums): if len(nums) == 0: return 1 n = len(nums) for i in range(n): while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]: temp = nums[i] - 1 nums[i], nums[temp] = nums[temp], nums[i] for i in range(n): if nums[i] != i + 1: return i + 1 return n + 1 </pre>	<pre> class Solution(object): def firstMissingPositive(self, nums): if not nums: return 1 n = len(nums) for i in range(n): if nums[i] <= 0 or nums[i] > n: nums[i] = n + 1 for num in nums: num = abs(num) if num > n: continue num -= 1 if nums[num] > 0: nums[num] = -num[num] for i in range(n): if nums[i] >= 0: return i + 1 return n + 1 </pre>	<pre> class Solution(object): def firstMissingPositive(self, nums): if not nums: return 1 n = len(nums) for i in range(n): while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]: nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1] for i in range(n): if nums[i] != i + 1: return i + 1 return n + 1 </pre>	<pre> class Solution(object): def firstMissingPositive(self, nums): if not nums: return 1 n = len(nums) for i in range(n): if nums[i] <= 0 or nums[i] > n: nums[i] = 1 for i in range(n): a = abs(nums[i]) if a == n: nums[0] = - abs(nums[0]) else: nums[a] = - abs(nums[a]) for i in range(1, n): if nums[i] > 0: return i if nums[0] > 0: return n return n + 1 </pre>
 Status: fail  Runtime: Fail	 Status: passed  Runtime: 0.00075126 s	 Status: passed  Runtime: 0.00087070s	 Status: passed  Runtime: 0.00073242 s	 Status: fail  Runtime: Fail

Figure 9: Qualitative analysis for Case 3 (first-missing-positive).