LEARNING TO GENERATE EFFICIENT CODE WITH PRIVILEGED EFFICIENCY CONTEXT AND AST-GUIDED SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have achieved remarkable success in code generation, yet functionally correct programs often present sub-optimal inefficiency in runtime efficiency and memory usage. Recently, there are some method have attempted to alleviate this problem. While effective, these methods only provide sparse feedback at the program level after generation is complete, making it difficult to provide fine-grained supervision during generation. To address this limitation, we propose EffiCode-OPD, an on-policy distillation (OPD)-based framework for efficient code generation to provide efficiency-aware token-level supervision during training. Specifically, EffiCode-OPD samples on-policy rollouts from the student model and uses a stronger teacher model to provide token-level supervision conditioned on an efficiency context that includes reference code, runtime, memory usage, and AST information. In addition, we design an AST-guided token weighting scheme that focuses the distillation objective on efficiency-relevant code segments. Experiments on representative code efficiency benchmarks show the effectiveness of EffCode-OPD for generating efficient code. For example, EffiCode-OPD reduces average runtime by up to 17.9% and memory usage by up to 19.4% relative to the strongest baseline on EFFIBENCH-X, while preserving functional correctness.

1 INTRODUCTION

Large language models (LLMs) have made substantial progress in code generation tasks Chen et al. (2021); Li et al. (2022; 2023); Nijkamp et al. (2023); Rozière et al. (2023); Hui et al. (2024). Through large-scale code pretraining and code instruction tuning, Code LLMs can now generate executable programs for program synthesis Chen et al. (2021); Nijkamp et al. (2023), code completion Wang et al. (2021); Li et al. (2023), and competitive programming Li et al. (2022); Rozière et al. (2023). However, existing evaluation and training pipelines mainly emphasize functional correctness, namely whether generated code can pass test cases, while paying less attention to runtime and memory usage during execution Huang et al. (2024b); Du et al. (2024); Liu et al. (2024); Qiu et al. (2024); Waghjale et al. (2024); Qing et al. (2025). For real software systems, functionally correct but inefficient code can still cause high latency, resource waste, or failure to meet deployment requirements. Therefore, training LLMs to generate code that is both correct and efficient has become an important problem in code generation.

Recent works have studied the execution efficiency of LLM-generated code from both evaluation and optimization perspectives. For evaluation, benchmarks such as EffiBench Huang et al. (2024b), Mercury Du et al. (2024), EvalPerf Liu et al. (2024), ENAMEL Qiu et al. (2024), ECCO Waghjale et al. (2024), and Effibench-X Qing et al. (2025) systematically measure runtime and memory usage, showing that functionally correct code generated by LLMs still be much less efficient than human-written code. For optimization, PerfCodeGen Peng et al. (2025a) uses execution feedback to iteratively refine code during inference, PIE Shypula et al. (2024) learns to edit inefficient code into more efficient code, EffiLearner Huang et al. (2024a) analyzes code to locate costly regions and rewrites them with LLMs, and LLM-as-a-Critique Zhu et al. (2026) combines static analysis with LLM critiques to score candidate code.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Despite achieving improvements, the primary limitation of existing optimization methods is that efficiency feedback is typically only accessible after the entire program has been generated, through using runtime and memory as feedback signal. The signal remains inherently sparse and restricted to the program level. This coarseness creates a significant challenge for fine-grained supervision, as it is difficult to map a single efficiency score back to specific token-level decisions during the generation process. In contrast, on-policy distillation (OPD) samples rollouts from the student model, treats each generated prefix as a training state, and lets the teacher model supervise the next-token distribution at that state Gu et al. (2024); Agarwal et al. (2024). This provides denser supervision than program-level feedback and trains the student under a generation distribution close to inference time. However, directly applying OPD provides no efficiency-specific guidance, since it cannot to determine which generation decisions are related to efficiency.

To address this problem, we propose EffiCode-OPD, an OPD-based framework for efficient code generation. Given a programming problem, the student model first generates on-policy rollouts. We then execute the student-generated code and an efficient reference program to collect runtime and memory information, and use Abstract Syntax Tree (AST) Aho et al. (2006) analysis to compare structural differences between the student program and the reference program. Based on the reference program and these signals, we construct a privileged efficiency context for the teacher model, so that the teacher model can provide efficiency-aware token-level supervision on student-generated prefixes. Furthermore, we identify efficiency-relevant AST regions in the student program, align them with token spans, and weight the corresponding token-level KL terms in the AST-guided OPD objective. After training, the student model independently generates code at inference time, without access to the teacher model, execution feedback, static analysis, LLM critique, or candidate reranking. Figure 1 provides a comparison of the probability assigned by different methods to the next token from an efficient reference solution. RL-based optimization changes this probability only marginally, and vanilla OPD provides a moderate improvement through prefix-level supervision. EffiCode-OPD assigns substantially higher probability to efficient next tokens, suggesting that our method provides useful efficiency-specific guidance for token-level supervision.

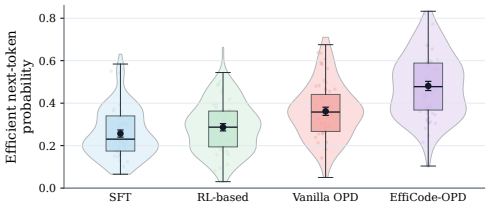


Figure 1: Efficient next-token probability across methods. For each student-generated prefix, we measure the probability assigned to the next token from an efficient reference solution.

We evaluate EffiCode-OPD on representative code generation benchmarks, including EffiBench Huang et al. (2024b), Effibench-X Qing et al. (2025), ENAMEL Qiu et al. (2024), and Mercury Du et al. (2024). Experimental results show the effectiveness of EffiCode-OPD. For example, on Effibench-X, it reduces average runtime by up to 17.9% and average memory usage by up to 19.4%, while preserving functional correctness and outperforming the strongest baseline in Pass@1.

Our contributions are summarized as follows.

- We propose EffiCode-OPD, the first framework that introduces OPD into efficient code generation. By learning from trajectories generated by the student model itself, EffiCode-OPD trains code LLMs to generate more efficient programs.
- We incorporate reference code, runtime, memory, and AST structural information into OPD, providing the teacher model with a privileged efficiency context and guiding distillation toward efficiency-relevant code segments through AST-guided token weighting.
- Experimental results on code generation benchmarks show that EffiCode-OPD improves the execution efficiency of generated code while preserving functional correctness and avoiding additional inference-time overhead.

2 Related Work

Code efficiency in LLM-generated code. Code LLMs have advanced code generation through large-scale pretraining and instruction tuning Chen et al. (2021); Li et al. (2022; 2023); Wang et al. (2021); Nijkamp et al. (2023); Rozière et al. (2023); Luo et al. (2024), with research largely focused on functional correctness. Benchmarks including EffiBench Huang et al. (2024b), Mercury Du

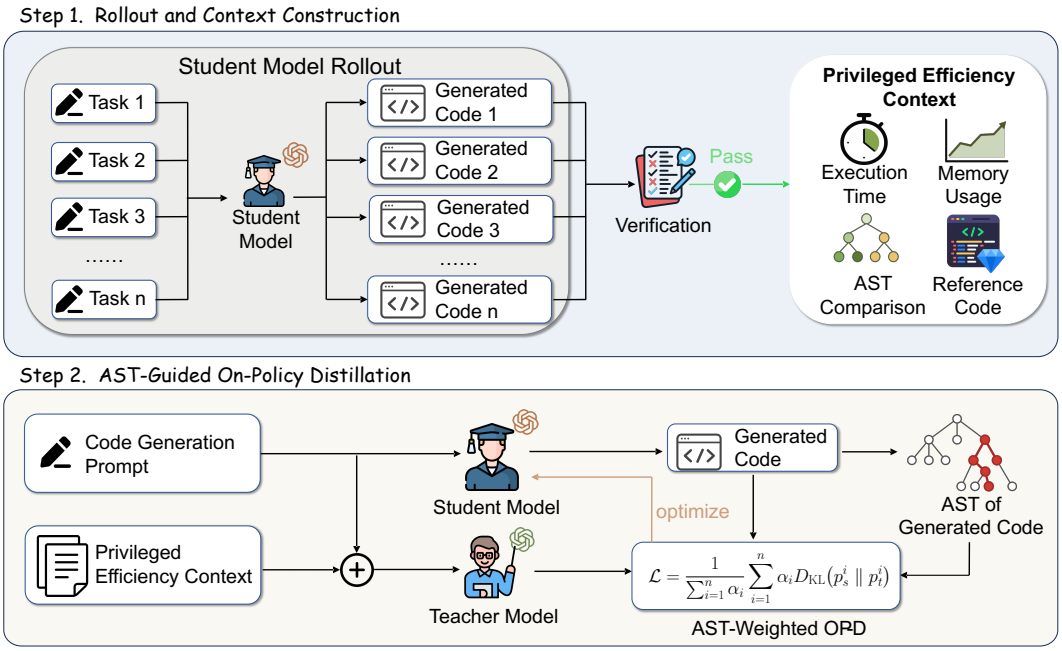


Figure 2: Overview of EffiCode-OPD. The framework consists of student on-policy rollouts, privileged efficiency context construction, and AST-guided token weighting for OPD training.

et al. (2024), EvalPerf Liu et al. (2024), ENAMEL Qiu et al. (2024), PIE Shypula et al. (2024), ECCO Waghjale et al. (2024), COFFE Peng et al. (2025b), and Effibench-X Qing et al. (2025) extend evaluation to runtime and memory usage. For optimization, PerfCodeGen Peng et al. (2025a) uses iterative execution feedback, PIE Shypula et al. (2024) learns efficient code edits, EffiLearner Huang et al. (2024a) uses profiling to rewrite costly code, and LLM-as-a-Critique Zhu et al. (2026) scores candidates through static analysis and LLM critiques. EffiCode-OPD internalizes efficiency signals through dense token-level supervision on student-generated code, improving execution efficiency without additional inference overhead.

On-policy distillation. On-policy distillation (OPD) reduces exposure bias by providing teacher supervision for each token on student-generated trajectories. MiniLLM Gu et al. (2024) formulates OPD with a reverse KL objective that favors high-likelihood regions of the teacher distribution, GKD Agarwal et al. (2024) unifies on-policy and off-policy distillation, and Learning Beyond Teacher Yang et al. (2026) connects OPD to dense KL-constrained RL and reward extrapolation. Extensions cover LLM post-training Lu & Lab (2025), black-box teachers Ye et al. (2026a), and self-distillation using privileged information such as ground-truth solutions or execution feedback Ding (2026); Hübotter et al. (2026); Zhao et al. (2026). On-Policy Context Distillation (OPCD) Ye et al. (2026b) internalizes in-context knowledge into model parameters, while EffiCode-OPD applies OPD to improving the execution efficiency of generated code.

3 METHOD

In this section, we present EffiCode-OPD, an OPD-based framework for improving code execution efficiency. As shown in Figure 2, we first sample on-policy rollouts from the student model and execute the generated code to verify its functional correctness. For code that passes the tests, we further execute the efficient reference code, record the runtime and memory usage of both programs, and compare their AST-level program structures. We then combine these execution profiles and structural differences into a privileged efficiency context that is provided only to an external teacher model. Second, during training, the teacher model uses this privileged context to provide token-level supervision on the student-generated prefixes. To further focus on efficiency optimization, we map efficiency-relevant AST regions to the corresponding token spans in the student code and weight the

distillation loss on these tokens in the OPD objective. Finally, at inference time, the student model generates code from the original problem statement alone, without requiring any additional efficient references, execution information, or access to the teacher model.

3.1 PRELIMINARIES

Let π_θ denote a code-generating language model parameterized by θ. Given a programming problem x, the model generates a sequence of code tokens $y = (y_1, y_2, \ldots, y_L)$ autoregressively:

$$\pi_\theta(y \mid x) = \prod_{l=1}^{L} \pi_\theta(y_l \mid x, y_{<l}),$$

where $y_{<l}$ denotes the prefix before token y_l. During standard inference, the model iteratively samples each token from the next-token distribution and feeds the sampled token back into the model to generate the following token until a stopping criterion is met.

3.2 ON-POLICY ROLLOUTS WITH EFFICIENCY CONTEXT

Execution-based profiling. For each programming problem x, we sample K on-policy rollouts from the student model π_s using temperature sampling with temperature $\tau = 1.0$. In our experiments, we set $K = 5$. Each rollout is a generated program $y^{(k)} = (y_1^{(k)}, \ldots, y_{n_k}^{(k)})$ produced under the same input format at inference, where $y^{(k)} \sim \pi_s^\tau(\cdot \mid x)$, $\tau = 1.0$, and $k = 1, \ldots, K$.

We execute each rollout on the test cases and keep only the programs that pass all tests. Failed rollouts are not used for efficiency distillation because execution efficiency is meaningful only for functionally correct programs. For each correct rollout y, we compare it with an efficient reference program r for the same problem. If no correct reference is more efficient than the student rollout, the rollout is excluded from OPD training.

We record the reference program together with the runtime and memory usage of both the student rollout and the reference program. Both programs are measured under the same execution environment and test suite. We use repeated execution with median runtime and peak memory usage to reduce measurement noise. Let t_y and m_y denote the runtime and memory usage of the student rollout y, and let t_r and m_r denote those of the reference program r. We define the execution profile as $p = (r, t_y, m_y, t_r, m_r)$.

AST structural contrast. While runtime and memory usage can reflect the efficiency of generated code, they lack the granularity to pinpoint inefficient code segments. To bridge this gap, we perform a structural comparison between the Abstract Syntax Trees (ASTs) of the student rollout and the reference. The AST analysis enumerates efficiency-related structural differences between the generated code and the reference code, including loop structures, function calls, data structure usage, recursion, caching, dynamic programming states, and repeated computation. These structural differences connect the observed execution behavior with concrete program-level patterns. The structural comparison procedure and its heuristic detection rules are specified in Appendix G.

Privileged efficiency context. Let a denote the AST structural contrast between the student rollout y and the reference program r. For each correct rollout, we construct the privileged efficiency context for the teacher model by combining the execution profile p and the AST structural contrast a, denoted as $c = (p, a)$.

The context c is provided only to the external teacher model during training and is never used as input to the student model. This design avoids a mismatch between training and inference, since the student model cannot access reference programs, execution results, or AST comparisons at inference time. In practice, c is serialized into a structured textual prompt for the teacher model, including the reference code, execution profile, and AST-level contrast. EffiCode-OPD combines on-policy training on student-generated prefixes with privileged conditioning of the teacher distribution, as detailed in Appendix C.2.

3.3 AST-GUIDED OPD TRAINING

After obtaining correct on-policy rollouts and their efficiency contexts, we train the student model with OPD, aiming to internalize the teacher's efficiency-aware insights into the student model through a fine-grained distillation objective. The student model and the teacher model receive different information during training. At token position i, the student model predicts the next-token distribution using only the original problem x and its own generated prefix $y_{<i}$, written as $p_s^i = \pi_s(\cdot \mid x, y_{<i})$. Besides the problem x, the teacher model predicts the next-token distribution on the privileged efficiency context c, written as $p_t^i = \pi_t(\cdot \mid x, y_{<i}, c)$.

On-policy distillation objective. Following OPD, we train the student on prefixes from its own generated rollout rather than on fixed reference trajectories. For a correct rollout $y = (y_1, \ldots, y_n)$, the token-level distillation loss at position i is defined by the reverse KL divergence from the student distribution to the teacher distribution:

$$D_i = D_{\mathrm{KL}}(p_s^i \parallel p_t^i) = \sum_{v \in V} p_s^i(v) \left(\log p_s^i(v) - \log p_t^i(v)\right).$$

Because the prefix $y_{<i}$ comes from the student rollout, this objective provides supervision on states that the student actually visits during generation. The teacher model conditions on the efficiency context and therefore provides token-level guidance that reflects the reference program, runtime, memory, and structural behavior of the current rollout.

AST-guided token weighting. Considering that not all tokens contribute equally to a program's efficiency, we introduce a weighting scheme, which assigns higher weights to tokens with stronger influences on program efficiency, involved in control flow, data structure usage, function calls, and repeated computation. In this paper, we analyze AST of the program to locate efficiency-relevant source regions in the student-generated program and increase the training weight of tokens aligned with these regions.

Let G_y denote the set of efficiency-relevant AST regions identified in the student rollout y. We construct G_y by parsing the student program into an AST and selecting nodes that correspond to efficiency-related code structures, such as loop structures, repeated function calls, recursion, caching structures, data structure usage, and repeated computation. After aligning source spans to token spans, we assign a token weight α_i to each token y_i:

$$\alpha_i = \begin{cases} \gamma, & \text{if token } y_i \text{ overlaps with a region in } G_y, \\ 1, & \text{otherwise,} \end{cases}$$

where $\gamma > 1$ is a hyperparameter controlling the strength of AST-guided weighting. The source-region selection and token-alignment rules are detailed in Appendix G.

Final training objective. Combining OPD with AST-guided token weighting, the loss for a correct rollout y with context c is normalized by the sum of token weights, so that AST-guided weighting changes the relative contribution of tokens within a rollout without changing the overall scale of the rollout-level loss:

$$\mathcal{L}(x, y, c) = \frac{1}{\sum_{i=1}^{n} \alpha_i} \sum_{i=1}^{n} \alpha_i D_{\mathrm{KL}}(p_s^i \parallel p_t^i).$$

We apply this loss to the subset of student rollouts that pass all tests and have a valid privileged efficiency context. Denoting this subset as $\mathcal{D}_{\text{eff}} = \{(x, y, c)\}$, where each context c contains a valid execution profile and AST structural contrast, the overall training objective is

$$\mathcal{L}_{\text{EffiCode-OPD}} = \mathbb{E}_{(x,y,c) \in \mathcal{D}_{\text{eff}}} \left[\mathcal{L}(x, y, c)\right].$$

Since this objective is applied only to correct rollouts with valid efficiency contexts, the distillation signal is constructed from executable programs while focusing updates on efficiency improvement.

3.4 INFERENCE FOR EFFICIENT CODE GENERATION

At inference time, given a new programming problem, the student model autoregressively generates a complete program without access to the external teacher model, efficient reference programs,

test execution results, runtime or memory measurements, or AST analysis. Let x' denote a new programming problem at inference time. Let $\hat{y}$ denote the program generated by the trained student model. The inference process is written as $\hat{y} \sim \pi_s(\cdot \mid x')$.

4 EXPERIMENT

4.1 EXPERIMENT SETTINGS

Datasets and Models. We conduct experiments on four representative code efficiency benchmarks. **EffiBench** Huang et al. (2024b) contains 1,000 efficiency-critical LeetCode problems with human-written solutions and generated test cases. **Mercury** Du et al. (2024) consists of 1,889 Python programming tasks from LeetCode, each equipped with test case generators and runtime-based difficulty annotations. **Effibench-X** Qing et al. (2025) contains 623 complex programming problems collected from multiple competitive programming platforms. **ENAMEL** Qiu et al. (2024) contains 142 HumanEval-style tasks with expert-optimized reference solutions and test generators designed to distinguish different efficiency levels. We split each dataset into training and test sets at an approximately 5:1, with details in Appendix C.1.

Baselines We compare with **SFT**, which learns from efficient reference solutions, and **PPO** Schulman et al. (2017), for which we define a composite reward based on functional correctness, runtime, and memory efficiency. For inference-time optimization, **EffiLearner** Huang et al. (2024a) rewrites code using execution profiles, **PerfCodeGen** Peng et al. (2025a) iteratively refines code with execution feedback, and **LLM-as-Critique** Zhu et al. (2026) selects efficient candidates through LLM critiques.

Base Models. We use four open-source base-model families to evaluate all methods, including Code Llama Rozière et al. (2023), Qwen3 Yang et al. (2025), Gemma3 Gemma Team (2025), and Qwen2.5-Coder Hui et al. (2024). For each model family, we select the largest evaluated model as the teacher and use the smaller variants as students. Specifically, CodeLlama-34B supervises CodeLlama-7B and 13B, Qwen3-32B supervises Qwen3-8B and 14B, Gemma3-27B supervises Gemma3-12B, and Qwen2.5-Coder-32B supervises Qwen2.5-Coder-7B and 14B, with teacher performance reported in Appendix H.

Evaluation Metrics. Following prior works on efficient code generation Huang et al. (2024b); Qing et al. (2025); Huang et al. (2024a), we evaluate functional correctness with **Pass@1** Chen et al. (2021) and execution efficiency with runtime and memory metrics. We compute Pass@1 over the full test set. For each benchmark and base model, we report **Execution Time (ET)** in seconds, **Max Memory Usage (MU)** in GB, and their dimensionless normalized variants **NET** and **NMU** over the programs generated by each method that pass all test cases. Detailed metric definitions are provided in Appendix D. For benchmarks without released memory annotations, we measure memory usage by running generated programs and reference solutions in the same isolated environment. Lower ET, NET, MU, and NMU indicate better efficiency, while higher Pass@1 indicates better correctness.

Implementation Details. We implement EffiCode-OPD using student on-policy rollouts and token-level supervision from a frozen teacher conditioned on reference code, runtime, memory, and AST information. We use dataset-provided reference code when available, otherwise sampling multiple teacher-generated programs and selecting the correct candidate with the best measured efficiency. The effect of reference-code quality is analyzed in Appendix K, and prompt templates are provided in Appendix M. Training and teacher inference use bfloat16 on a SLURM-managed cluster with eight NVIDIA A100 80GB GPUs per node, while execution performance is measured in identical isolated CPU-only environments across methods. Additional sampling, training, and post-processing details are provided in Appendix C.

4.2 MAIN RESULTS

Table 1 shows that EffiCode-OPD improves both generation correctness and execution efficiency across model families and benchmarks. Compared with the strongest baseline for each metric, EffiCode-OPD reduces ET, NET, MU, and NMU in all reported settings, while also increasing

Model	Method	EffiBench					Effibench-X				
		ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
Code-Llama-7B	Few-shot	1.83	1.52	1.21	1.79	10.8	1.92	1.76	1.33	1.68	8.7
	SFT	1.79	1.51	1.18	1.77	13.7	1.89	1.77	1.31	1.66	9.3
	PPO	1.52	1.34	0.99	1.45	19.8	1.57	1.51	1.14	1.47	13.4
	EffiLearner	1.65	1.32	1.21	1.52	12.9	1.71	1.67	1.30	1.57	10.2
	PerfCodeGen	1.61	1.23	1.08	1.33	18.8	1.63	1.50	1.29	1.52	11.3
	LLM-as-Critique	1.22	1.06	0.93	1.02	19.5	1.23	1.29	1.03	1.19	12.8
	EffiCode-OPD (Ours)	**1.03**	**0.89**	**0.73**	**0.72**	**22.5**	**1.01**	**1.12**	**0.83**	**1.03**	**14.5**
	Gain(Δ)	↓15.6%	↓16.0%	↓21.5%	↓29.4%	↑13.6%	↓17.9%	↓13.2%	↓19.4%	↓13.4%	↑8.2%
Qwen3-8B	Few-shot	1.72	1.49	1.31	1.29	12.3	1.93	1.78	1.41	1.62	11.2
	SFT	1.70	1.44	1.33	1.22	14.3	1.97	1.77	1.43	1.64	13.6
	PPO	1.48	1.22	1.19	1.03	17.3	1.72	1.53	1.26	1.44	17.4
	EffiLearner	1.62	1.35	1.28	1.17	15.6	1.74	1.72	1.54	1.55	16.9
	PerfCodeGen	1.51	1.26	1.17	1.08	17.7	1.68	1.62	1.37	1.36	17.2
	LLM-as-Critique	1.09	1.02	0.89	0.93	17.8	1.16	1.09	0.98	1.13	17.9
	EffiCode-OPD (Ours)	**0.91**	**0.93**	**0.73**	**0.84**	**19.3**	**1.05**	**0.95**	**0.87**	**1.02**	**19.8**
	Gain(Δ)	↓16.5%	↓8.8%	↓18.0%	↓9.7%	↑8.4%	↓9.5%	↓12.8%	↓11.2%	↓9.7%	↑10.6%
Qwen3-14B	Few-shot	1.53	1.31	1.20	1.09	18.3	1.72	1.54	1.23	1.51	19.2
	SFT	1.51	1.29	1.23	1.11	21.3	1.71	1.51	1.24	1.48	20.1
	PPO	1.32	1.17	1.12	1.02	24.8	1.66	1.43	1.12	1.23	23.4
	EffiLearner	1.42	1.25	1.10	1.03	20.5	1.73	1.52	1.17	1.33	20.3
	PerfCodeGen	1.36	1.18	1.03	0.95	25.3	1.58	1.31	1.08	1.17	24.6
	LLM-as-Critique	0.86	0.83	0.79	0.83	25.1	1.37	0.81	0.92	1.06	24.3
	EffiCode-OPD (Ours)	**0.72**	**0.76**	**0.64**	**0.77**	**28.4**	**1.03**	**0.72**	**0.74**	**0.94**	**26.9**
	Gain(Δ)	↓16.3%	↓8.4%	↓19.0%	↓7.2%	↑12.3%	↓24.8%	↓11.1%	↓19.6%	↓11.3%	↑9.3%
Qwen2.5-Coder-7B	Few-shot	1.68	1.46	1.28	1.26	15.2	1.89	1.75	1.43	1.66	17.8
	SFT	1.70	1.45	1.29	1.28	16.7	1.91	1.74	1.45	1.68	19.2
	PPO	1.54	1.23	1.16	1.08	20.8	1.83	1.52	1.13	1.34	22.3
	EffiLearner	1.67	1.38	1.23	1.19	18.5	1.82	1.63	1.37	1.55	21.8
	PerfCodeGen	1.63	1.33	1.21	1.15	19.3	1.89	1.62	1.28	1.43	21.7
	LLM-as-Critique	1.38	1.18	1.08	1.03	20.4	1.69	1.43	1.02	1.18	23.8
	EffiCode-OPD (Ours)	**1.18**	**1.03**	**0.92**	**0.91**	**22.5**	**1.43**	**1.21**	**0.95**	**1.07**	**25.4**
	Gain(Δ)	↓14.5%	↓12.7%	↓14.8%	↓11.7%	↑8.2%	↓15.4%	↓15.4%	↓6.9%	↓9.3%	↑6.7%
Qwen2.5-Coder-14B	Few-shot	1.51	1.34	1.26	1.21	22.3	1.74	1.63	1.51	1.59	22.4
	SFT	1.48	1.36	1.24	1.22	24.5	1.72	1.65	1.47	1.58	23.9
	PPO	1.22	1.07	1.13	1.02	28.4	1.46	1.36	1.32	1.23	27.4
	EffiLearner	1.38	1.26	1.15	1.09	25.6	1.65	1.48	1.23	1.43	25.8
	PerfCodeGen	1.38	1.24	1.17	1.07	26.3	1.58	1.52	1.16	1.38	26.4
	LLM-as-Critique	1.11	1.05	0.93	0.96	27.3	1.43	1.23	1.25	1.17	27.3
	EffiCode-OPD (Ours)	**1.03**	**0.96**	**0.83**	**0.85**	**30.1**	**1.23**	**1.15**	**1.08**	**1.08**	**29.6**
	Gain(Δ)	↓7.2%	↓8.6%	↓10.8%	↓11.5%	↑6.0%	↓14.0%	↓6.5%	↓6.9%	↓7.7%	↑8.0%

Table 1: Main results on representative models and benchmarks. ET is measured in seconds, MU is measured in GB, and NET and NMU are dimensionless. Efficiency metrics are averaged over the programs generated by each method that pass all test cases, while Pass@1 is evaluated on the full test set. Lower is better for ET, NET, MU, and NMU, while higher is better for Pass@1. Our method is shaded, Gain(Δ) rows are highlighted, and the best result in each model is shown in **bold**. Red percentages report changes relative to the best baseline for each metric.

Pass@1. For example, on EffiBench, Code-Llama-7B reduces ET, NET, MU, and NMU by 15.6%, 16.0%, 21.5%, and 29.4%, respectively, and improves Pass@1 by 13.6%. On Effibench-X, Qwen3-14B reduces ET, NET, MU, and NMU by 24.8%, 11.1%, 19.6%, and 11.3%, respectively, and improves Pass@1 by 9.3%. The full results reported in Appendix N show the same trend across the remaining model and benchmark settings.

The improvements remain visible for stronger student models, including Qwen3-14B and Qwen2.5-Coder-14B, which indicates that EffiCode-OPD does not only benefit low-capacity models. Compared with LLM-as-Critique, the strongest feedback-based baseline in almost all settings, EffiCode-OPD achieves better efficiency and accuracy. The gains also cover both runtime and memory metrics, showing that the efficiency improvement is not limited to a single resource dimension.

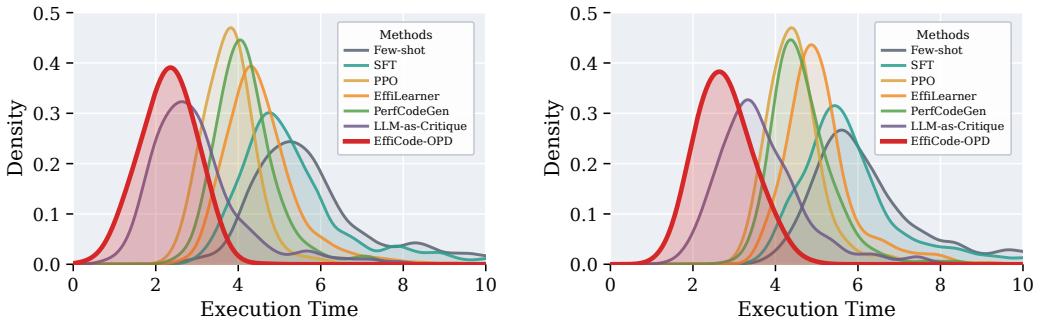
In conclusion, our method integrates on-policy distillation with privileged prompting for the teacher model guided by correctness and efficiency contexts, improving code efficiency while maintaining correctness.

4.3 Analysis

Ablation Study. Table 2 supports the effectiveness of both privileged efficiency context and AST-guided token weighting. Removing the privileged efficiency context increases NET from 0.89 to 1.14 and NMU from 0.72 to 1.09. Removing AST-guided token weighting while retaining the full

Variant	Privileged Efficiency Context				AST-Guided Token Weighting	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
	Ref. Code	Runtime	Memory	AST Struct.						
Full EffiCode-OPD	✓	✓	✓	✓	✓	1.03	0.89	0.73	0.72	22.5
Vanilla OPD	×	×	×	×	×	1.31	1.15	1.02	1.11	19.6
w/o privileged efficiency context	×	×	×	×	✓	1.29	1.14	1.00	1.09	20.0
w/o AST-guided token weighting	✓	✓	✓	✓	×	1.28	1.13	0.99	1.08	21.3
w/o reference code in prompt	×	✓	✓	✓	✓	1.26	1.12	0.97	1.06	22.1
w/o AST information in prompt	✓	✓	✓	×	✓	1.27	1.13	0.98	1.07	22.1
w/o runtime in prompt	✓	×	✓	✓	✓	1.25	1.11	0.78	0.77	22.3
w/o memory in prompt	✓	✓	×	✓	✓	1.07	0.94	1.00	1.08	22.4
w/o runtime + memory in prompt	✓	×	×	✓	✓	1.26	1.12	0.97	1.06	21.2

Table 2: Ablation study of EffiCode-OPD on Code-Llama-7B and EffiBench. All variants keep the OPD training framework and vary the privileged efficiency context used in prompting the teacher model or the AST-guided token weighting used in the student loss.



(a) EffiBench (b) Effibench-X

Figure 3: Execution time distributions of evaluated methods on EffiBench and Effibench-X.

context increases NET to 1.13 and NMU to 1.08. Pass@1 also decreases from 22.5 to 20.0 and 21.3, respectively.

The decomposed variants examine the information supplied to the teacher model. Removing either reference code or AST structural information increases all four efficiency metrics. Removing runtime measurements primarily affects runtime metrics, while removing memory measurements primarily affects memory metrics. Removing both measurements increases NET to 1.12 and NMU to 1.06. Additional ablation results are provided in Appendix F.

Generated Code Efficiency Distribution. We further analyze the execution time distribution of generated code across benchmark tasks. As shown in Figure 3, Few-shot and SFT produce more samples in slower execution-time regions, while feedback-based methods reduce this pattern to some extent. EffiCode-OPD shows a more concentrated distribution in shorter execution-time ranges on both EffiBench and Effibench-X, with fewer samples remaining in high execution-time regions. This distributional pattern is consistent with the average ET improvements in Table 1.

Training Curve Analysis. To analyze the advantage of token-level supervision, we compare the training dynamics of EffiCode-OPD against vanilla OPD, PPO, and SFT under the same training budget. We report task-level evaluation metrics, including Pass@1, ET, and MU, at different training checkpoints. This comparison characterizes how each method improves correctness and efficiency during training, including both convergence speed and training stability.

As shown in Figure 4, EffiCode-OPD achieves faster and more stable improvements in execution time and memory usage while maintaining competitive Pass@1. Compared with PPO, which relies on delayed program-level scalar feedback, EffiCode-OPD benefits from dense token-level guidance along on-policy trajectories. Compared with vanilla OPD, the improvement indicates that emphasizing AST-aligned efficiency-relevant tokens provides more informative supervision than treating all generated tokens uniformly.

Teacher Model Selection. We analyze how teacher selection affects EffiCode-OPD in efficient code generation. To isolate the effect of the teacher model, we keep the student model, training

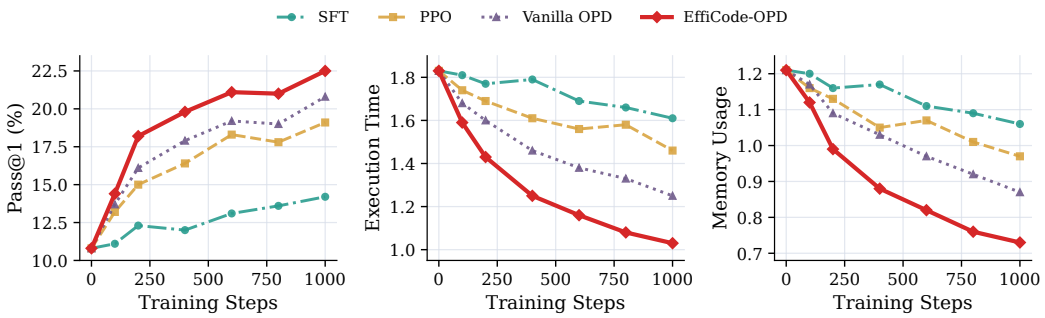


Figure 4: Training curves on EffiBench using Code-Llama-7B as the base model, comparing SFT, PPO, vanilla OPD, and EffiCode-OPD with metrics including Pass@1, ET, and MU.

Student Model	Teacher Model	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
Code-Llama-7B	Code-Llama-13B	1.15	1.01	0.86	0.91	21.3
	Code-Llama-34B	**1.03**	**0.89**	**0.73**	**0.72**	**22.5**
	Qwen2.5-Coder-14B	1.28	1.12	0.98	1.06	20.1
	Qwen2.5-Coder-32B	1.21	1.06	0.91	0.98	20.8
Qwen2.5-Coder-7B	Code-Llama-13B	1.45	1.25	1.10	1.13	20.4
	Code-Llama-34B	1.36	1.18	1.03	1.06	21.1
	Qwen2.5-Coder-14B	1.27	1.10	0.99	0.98	21.8
	Qwen2.5-Coder-32B	**1.18**	**1.03**	**0.92**	**0.91**	**22.5**

Table 3: Effect of teacher model family on EffiBench. The table keeps the student model fixed and compares same-family and cross-family teacher models.

procedure, and evaluation setting fixed, and only replace the teacher used to provide token-level supervision. We conduct this comparison on EffiBench using Code-Llama-7B and Qwen2.5-Coder-7B as student models, and compare teachers from the same model family and a different model family. To enable OPD between models from different families, we perform tokenizer alignment, with the detailed procedure provided in Appendix L. As shown in Table 3, teachers from the same model family consistently achieve better results than teachers from a different model family for both student models. For Code-Llama-7B, the Code-Llama teachers outperform the Qwen2.5-Coder teachers, while for Qwen2.5-Coder-7B, the Qwen2.5-Coder teachers give the best results. This shows that same-family teacher selection is more effective in our setting.

Within the same model family, larger teachers further improve the student model. For Code-Llama-7B, Code-Llama-34B outperforms Code-Llama-13B, and for Qwen2.5-Coder-7B, Qwen2.5-Coder-32B outperforms Qwen2.5-Coder-14B. These results indicate that, for our task, teacher selection benefits from both same-family pairing and larger teacher scale.

5 Conclusion

In this paper, we introduce EffiCode-OPD, an on-policy distillation framework for efficient code generation. During training, EffiCode-OPD provides the teacher model with privileged context consisting of reference efficient code, runtime, memory usage, and AST structural information, enabling the teacher model to produce efficiency-aware token-level supervision. Meanwhile, EffiCode-OPD uses AST structural analysis to identify key tokens related to code efficiency and strengthens the learning signal at these positions in the distillation objective. Experimental results show that EffiCode-OPD effectively improves the execution efficiency of generated code while preserving functional correctness. In future work, we plan to study larger teacher models, reduce the dependence on high-quality reference programs, and extend EffiCode-OPD to broader programming languages and more complex software-engineering tasks.

AI USE STATEMENT

In this work, we used generative AI tools to assist with and polish the writing, to support literature retrieval and the discovery of related work, and to aid research ideation and execution. All AI-assisted text, literature suggestions, research ideas, and outputs were critically reviewed and verified by the authors. The authors independently checked the accuracy and relevance of suggested references, validated AI-assisted research outputs, and revised the resulting content as necessary. We take full responsibility for the final content of this work, including all text, claims, citations, and artifacts produced with the aid of generative AI.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations*, 2024.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2006.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Ken Ding. Hdpo: Hybrid distillation policy optimization via privileged self-distillation. *arXiv preprint arXiv:2603.23871*, 2026.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37, 2024.

Gemma Team. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. In *International Conference on Learning Representations*, 2024.

Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie Zhang. Effilearner: Enhancing efficiency of generated code via self-optimization. In *Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37, 2024b.

Jonas Hübotter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, and Andreas Krause. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*, 2026.

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

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: May the source be with you! *Transactions on Machine Learning Research*, 2023.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*, 2024.

Kevin Lu and Thinking Machines Lab. On-policy distillation. Thinking Machines Lab: Connectionism, 2025. URL https://thinkingmachines.ai/blog/on-policy-distillation.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024.

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

A Paszke. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

Yun Peng, Akhilesh Deepak Gotmare, Caiming Xiong, Michael Lyu, and Doyen Sahoo. Perfcodegen: Improving performance of llm generated code with execution feedback. In *Proceedings of the 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering*, pp. 1–13, 2025a.

Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. Coffe: A code efficiency benchmark for code generation. *arXiv preprint arXiv:2502.02827*, 2025b.

Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, See-Kiong Ng, and Anh Tuan Luu. Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code. *arXiv preprint arXiv:2505.13004*, 2025.

Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous and high-standard benchmark. *arXiv preprint arXiv:2406.06647*, 2024.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024.

Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Wenkai Yang, Weijie Liu, Ruobing Xie, Kai Yang, Saiyong Yang, and Yankai Lin. Learning beyond teacher: Generalized on-policy distillation with reward extrapolation. *arXiv preprint arXiv:2602.12125*, 2026.

Tianzhu Ye, Li Dong, Zewen Chi, Xun Wu, Shaohan Huang, and Furu Wei. Black-box on-policy distillation of large language models. *arXiv preprint*, 2026a.

Tianzhu Ye, Li Dong, Xun Wu, Shaohan Huang, and Furu Wei. On-policy context distillation for language models. *arXiv preprint arXiv:2602.12275*, 2026b.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.

Yaowei Zheng, Richong Zhang, Junhao Zhang, YeYanhan YeYanhan, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, 2024.

Derui Zhu, Dingfan Chen, Jens Grossklags, Alexander Pretschner, Weiyi Shang, et al. More than just functional: Llm-as-a-critique for efficient code generation. *Advances in Neural Information Processing Systems*, 38:108690–108719, 2026.

A LIMITATIONS

This paper has the following limitations: First, due to hardware and computational constraints, we have not studied the effect of larger teacher models, such as 72B-scale or larger models. Second, our method relies on executable test cases, a stable profiling environment, and high-quality efficient reference programs during training. Therefore, constructing training data can be more difficult for tasks where tests are unavailable, execution is costly, or reference solutions are of insufficient quality.

B RESPONSIBLE RESEARCH DISCLOSURES

Compute resources. Student training and teacher-side scoring are conducted on a SLURM-managed cluster with NVIDIA A100 GPUs with 80GB memory. Code execution, profiling, and evaluation are run on isolated CPU-only nodes with identical software environments for all methods.

Broader impacts. EffiCode-OPD aims to improve the execution efficiency of LLM-generated code, which can reduce runtime, memory usage, and deployment cost for code-generation systems. At the same time, more capable code-generation models may still produce incorrect, insecure, or inefficient programs when used outside the evaluated setting. The method should therefore be used with standard software-engineering safeguards, including testing, profiling, code review, and security analysis before deployment.

Existing assets and licenses. This work uses existing public benchmarks and open model families, including EffiBench, Mercury, EFFIBENCH-X, ENAMEL, Code Llama, Qwen3, Gemma 3, and Qwen2.5-Coder. We cite the corresponding papers and use these assets for research evaluation under their released terms and licenses. We do not introduce a new benchmark dataset or release a new pretrained model as part of this submission.

LLM usage. LLMs are central to the proposed method and experiments. Student LLMs are used to generate code rollouts and final solutions, while teacher LLMs provide efficiency-aware token-level supervision and synthesize candidate reference programs when a dataset does not provide reference code. The synthesized reference candidates are filtered by correctness tests and measured execution efficiency before being used for training.

C IMPLEMENTATION DETAILS

C.1 DATA PARTITIONS AND VALIDATION

EffiBench contains 1,000 efficiency-critical LeetCode problems with human-written solutions and generated test cases. Mercury consists of 1,889 Python programming tasks from LeetCode, each equipped with test case generators and runtime-based difficulty annotations. Effibench-X contains 623 complex programming problems collected from multiple competitive programming platforms. ENAMEL contains 142 HumanEval-style tasks with expert-optimized reference solutions and test generators designed to distinguish different efficiency levels.

We partition each benchmark independently into a training pool and a held-out test set at an approximately 5:1 ratio, yielding 3,044 training-pool problems and 610 test problems in total. Table 4 reports the training pools before validation holdout and the held-out test sets. Cross-benchmark duplicate checks use problem-description text and reference-code AST structure to exclude overlap between training and test problems. Approximately 10% of each training pool, totaling approximately 304 problems, is reserved for hyperparameter selection and checkpoint evaluation. These validation problems are excluded from gradient updates. Training rollouts, correctness filtering, and efficiency contexts use only problems and associated assets from the remaining training partitions. Held-out test problems, their reference solutions, and their test cases are excluded from training and model selection. The main results are evaluated on the held-out test partitions, which are used only for final evaluation.

Benchmark	Total	Training pool	Test
EffiBench	1,000	833	167
Mercury	1,889	1,574	315
Effibench-X	623	519	104
ENAMEL	142	118	24
Total	3,654	3,044	610

Table 4: Problem-level dataset partitions. The training-pool counts include the validation holdout of approximately 10% per benchmark.

C.2 Distillation Setting

EffiCode-OPD combines on-policy distillation with privileged efficiency context. The student generates rollouts from the programming problem, and the teacher supervises the next-token distribution at the resulting student-generated prefixes. Reference code, execution measurements, and AST structural comparisons condition the teacher distribution during training. The student receives only the problem and its own prefix and is optimized using the AST-weighted distillation objective.

C.3 Rollouts and Execution

During training, the student model generates on-policy rollouts under nucleus sampling with $p = 0.95$. We use temperature $\tau = 1.0$ and sample $K = 5$ rollouts for each programming problem. Each rollout is generated from the standard code-generation prompt that contains only the original problem statement.

The teacher model is kept frozen and is used only to compute token-level supervision conditioned on the privileged efficiency context. For the main experiments, the teacher is selected from the same model family as the student and is the largest evaluated model in that family. Specifically, Code Llama 34B supervises Code Llama 7B and 13B, Qwen3 32B supervises Qwen3 8B and 14B, Gemma 3 27B supervises Gemma 3 12B, and Qwen2.5-Coder 32B supervises Qwen2.5-Coder 7B and 14B. The performance of teacher models is reported in Appendix H. The teacher input places the original problem and serialized privileged efficiency context before the student-generated prefix, which forms the final input tokens as specified in Appendix M. The student is always optimized on prefixes from its own rollout.

Student training and teacher-side scoring are conducted in bfloat16 on a SLURM-managed GPU cluster with 8 NVIDIA A100 GPUs with 80GB memory per node, connected through NVLink. Training and inference for open source models are implemented with PyTorch Paszke (2019), Huggingface Transformers Wolf et al. (2020), and LLaMA-Factory Zheng et al. (2024). We use distributed data parallel training with gradient accumulation to support large-batch OPD updates, and fix random seeds for rollout sampling, data ordering, and stochastic evaluation runs.

For code execution evaluation, generated programs and reference solutions are run in isolated Docker containers on dedicated CPU-only nodes with identical software environments. Runtime and memory are measured on the same hardware for all methods, and each program is executed multiple times with a warm-up run before recording the median runtime and peak memory usage to reduce system-level noise. Only rollouts that pass all test cases are used for efficiency distillation.

For each correct rollout, we execute both the student program and the selected reference program, record their runtime and peak memory usage, and construct the execution profile $p = (r, t_y, m_y, t_r, m_r)$. If the dataset provides reference code, we use the provided reference as r. Otherwise, we sample five candidate programs from the teacher model, keep only candidates that pass all tests, profile the correct candidates, and select the one with the best measured efficiency as the reference. If the selected reference is not more efficient than the student rollout, the rollout is excluded from efficiency-context construction. The effect of reference-code quality is analyzed in Appendix K, and an extension using student-generated references is discussed in Appendix K.1.

We parse both the student and reference programs into ASTs and compare their efficiency-relevant structures to construct the AST structural contrast used in the teacher-side privileged context. For

AST-guided token weighting, we parse the student program AST, select student-side source regions corresponding to efficiency-related structures, and map these regions back to token spans using source offsets. Only tokens overlapping with these spans receive the larger KL weight in the OPD loss. Before evaluation, we apply a lightweight post-processing step to extract executable code from model responses without modifying the code content.

C.4 Optimization and Baseline Configurations

SFT. SFT uses efficient reference solutions from the training partitions after validation holdout. The learning rate is 2×10^{-5} with cosine decay, a 3% warmup fraction, and weight decay of 0.01. The effective global batch size is 64, using eight A100 GPUs, one example per device, and eight gradient accumulation steps.

PPO. PPO samples five rollouts per problem with temperature $\tau = 1.0$ and nucleus probability $p = 0.95$. Its composite reward incorporates functional correctness, runtime, and memory usage. The KL regularization coefficient is 0.02. The actor and critic learning rates are 5×10^{-6} and 10^{-5}, respectively. The PPO clipping parameter is 0.2, the generalized advantage estimation parameter is 0.95, and the effective global batch size is 64.

EffiLearner and PerfCodeGen. Each method starts with five candidate programs sampled at temperature $\tau = 0.8$ and uses at most three rounds of execution feedback and code refinement. Refinement stops when the revised program passes all tests and improves both runtime and memory relative to the preceding round, or when the three-round limit is reached.

LLM-as-Critique. The method samples eight candidates per problem with temperature $\tau = 0.8$ and nucleus probability $p = 0.95$. The critique model is the corresponding large model from the same family, such as Code Llama 34B or Qwen3 32B. It assigns a combined correctness and execution-efficiency score from 1 to 10 under the critique prompt. The highest-scoring candidate among those passing all tests is selected as the final output.

EffiCode-OPD. The learning rate is 2×10^{-5} for 7B and 8B students and 10^{-5} for 12B, 13B, and 14B students. We use cosine decay with a 3% warmup fraction. The effective global batch size is 64 across eight A100 GPUs with eight gradient accumulation steps. The AST token weight is $\gamma = 2.0$.

Shared settings and computational budgets. All methods use the same data partitions, a maximum generation length of 1,024 tokens, and identical Docker-isolated CPU environments for execution measurements. The candidate and refinement budgets above specify the additional generation and feedback calls used by the inference-time baselines. EffiCode-OPD uses the teacher, reference programs, and AST analysis during training and returns a single student generation at inference. End-to-end inference processing times, including candidate generation, feedback, critique, and selection, are reported in Appendix E.

D Evaluation Metric Definitions

We provide the detailed definitions of the efficiency metrics used in our experiments. For a method a, let $\mathcal{C}_a$ denote the set of test problems for which its generated program passes all test cases, with $N_a = |\mathcal{C}_a|$. For each problem $i \in \mathcal{C}_a$, T_i denotes execution time in seconds and M_i denotes peak memory usage in GB over the test suite, while T_i^{ref} and M_i^{ref} denote the corresponding measurements of the reference solution under the same environment. Pass@1 is computed over the full test set for each method.

Execution Time (ET). Execution Time measures the average runtime in seconds of correct generated programs:

$$\text{ET}_a = \frac{1}{N_a} \sum_{i \in \mathcal{C}_a} T_i.$$

Normalized Execution Time (NET). Normalized Execution Time measures runtime relative to the corresponding reference solution:

$$\mathrm{NET}_a = \frac{1}{N_a} \sum_{i \in \mathcal{C}_a} \frac{T_i}{T_i^{\mathrm{ref}}}.$$

A lower NET indicates that generated programs are closer to, or faster than, the reference solutions in runtime.

Max Memory Usage (MU). Max Memory Usage measures the average peak memory consumption in GB of correct generated programs:

$$\mathrm{MU}_a = \frac{1}{N_a} \sum_{i \in \mathcal{C}_a} M_i.$$

Normalized Max Memory Usage (NMU). Normalized Max Memory Usage measures peak memory consumption relative to the corresponding reference solution:

$$\mathrm{NMU}_a = \frac{1}{N_a} \sum_{i \in \mathcal{C}_a} \frac{M_i}{M_i^{\mathrm{ref}}}.$$

A lower NMU indicates better memory efficiency relative to the reference solutions.

E Inference Computation Overhead

Model	EffiLearner	PerfCodeGen	LLM-as-Critique	EffiCode-OPD
Code-Llama-7B	3212.49	4144.29	486.73	**28.39**
Qwen3-8B	3009.78	3987.53	452.18	**26.79**
Qwen3-14B	5316.53	6985.29	873.64	**58.47**
Qwen2.5-Coder-7B	6242.98	8288.58	769.35	**56.78**
Qwen2.5-Coder-14B	17145.45	22180.94	2146.82	**101.34**

Table 5: Median inference processing time in seconds on EffiBench. The reported time includes all inference-time computation required to return the final solution. EffiCode-OPD requires only a single student-model generation at inference time because execution feedback, AST structural information, and teacher-side guidance are used only during training.

Inference-time computation is an important practical factor for efficiency-oriented code generation methods. We measure inference processing time on EffiBench in seconds. For each model-method pair, we run the complete inference pipeline on every benchmark task and measure the wall-clock time from receiving the programming problem to returning the final generated solution. We repeat this measurement three times and report the median total processing time over the repeated runs. The measurement includes all inference-time model calls, candidate generation, execution feedback, static analysis, critique, refinement, and reranking steps required by each method.

EffiCode-OPD uses execution measurements, memory measurements, AST structural information, and teacher-side token distributions only during training. After training, the student model directly generates one solution without execution feedback, static analysis, teacher-model querying, or reranking. In contrast, inference-time feedback methods such as EffiLearner, PerfCodeGen, and LLM-as-Critique require additional inference-time computation to profile, critique, refine, or rerank generated programs.

Table 5 shows that EffiCode-OPD has substantially lower inference-time computation than inference-time feedback baselines. LLM-as-Critique is less expensive than methods that repeatedly profile or refine candidate programs, but it still requires additional critique and selection steps during inference. EffiLearner and PerfCodeGen have higher processing time because they rely on profiling, execution feedback, and iterative optimization at deployment time. These results indicate that EffiCode-OPD shifts the use of privileged efficiency signals to training and avoids extra inference-time components.

F Additional Ablation Study

Variant	Privileged Efficiency Context				AST-Guided Token Weighting	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
	Ref. Code	Runtime	Memory	AST Struct.						
Full EffiCode-OPD	✓	✓	✓	✓	✓	**1.03**	**0.89**	**0.73**	**0.72**	**22.5**
Only reference code prompt	✓	×	×	×	×	1.34	1.18	1.05	1.14	19.0
Only runtime prompt	×	✓	×	×	×	1.42	1.27	1.15	1.25	18.0
Only memory prompt	×	×	✓	×	×	1.43	1.28	1.14	1.24	18.1
Only AST structure prompt	×	×	×	✓	×	1.40	1.25	1.12	1.22	18.3
Only AST token weighting	×	×	×	×	✓	1.35	1.19	1.06	1.15	18.9

Table 6: Additional single-component ablation study for Code-Llama-7B on EffiBench. Each variant keeps only one source of efficiency guidance while using the same OPD training pipeline.

We further evaluate whether each component can work effectively when used alone. All variants use the same OPD training pipeline on Code-Llama-7B on EffiBench, but keep only one source of efficiency guidance.

Table 6 shows that using any single component alone is much weaker than the full method. Among the single-component variants, using only the reference-code prompt and using only AST-guided token weighting give relatively better results than using only runtime, memory, or AST-structure prompting. However, these single-component variants are still worse than the ablation variants in Table 2, where the model keeps most components and removes only one part. This indicates that the individual signals are insufficient by themselves, and that the full method benefits from combining reference code, execution profiles, AST structure, and token-level focus.

G Details about AST

This section provides additional details on how AST information is used in EffiCode-OPD. There are two different AST-derived signals. The first is an AST structural profile that compares the student-generated program with the efficient reference program and is included in the privileged context for the teacher. The second is a token-level AST weighting map that assigns larger loss weights to efficiency-relevant tokens in the student rollout. These two signals are related because they are both derived from AST analysis, but they play different roles in the training pipeline.

Student and reference ASTs. Given a student rollout y, we first parse the generated code into a student AST A_y. We also parse the efficient reference program r into a reference AST A_r. The two ASTs expose syntactic structures such as loops, nested branches, function calls, container operations, and indexed updates. We compare the parsed structures to describe differences between the student and reference programs for the teacher context. For token weighting, we select source regions from the student AST using the efficiency-related structural categories described below.

Code structures covered by AST analysis. The analysis uses a fixed set of syntactic rules to identify structures potentially relevant to runtime and memory usage. Each detected occurrence records its structural category, enclosing function, loop nesting depth, and source interval. The rules cover the following categories.

1. **Loops and branches.** Record loop and conditional nodes, their condition or iterator expressions, and maximum loop nesting depth. For example, a loop containing another loop contributes a nesting depth of two.
2. **Calls and direct recursion.** Record call expressions and their syntactically identifiable targets. Mark a direct recursive call when a named call targets its enclosing named function. Calls inside loops retain their enclosing loop depth.
3. **Containers and sorting.** Record container literals, comprehensions, explicit list, dictionary, set, and tuple constructors, membership tests, indexed access, slicing, and named sorting calls. Container mutation candidates include calls named `append`, `extend`, `insert`, `add`, and `update`.
4. **Caching and indexed state updates.** Record explicit `cache` and `lru_cache` decorators, indexed assignments, and membership guards followed by returning the corresponding indexed

value. A matching indexed write within the same function provides a candidate cache pattern. Indexed updates are retained as structural evidence without assigning an algorithmic label such as dynamic programming solely from the node type.

5. **Repeated expressions.** Within each function, group expression subtrees with identical syntax after removing source-location metadata and ignoring formatting. Preserve identifiers, operators, and literal values, and exclude standalone names and constants. Groups with multiple occurrences are recorded as repeated-expression candidates.

AST structural profile for the teacher. Structural contrast is computed by comparing category-level profiles of the two complete programs. The procedure takes the student AST A_y and reference AST A_r as input and produces the textual contrast a.

1. Traverse each AST with the same rules and collect the occurrence records described above.
2. Aggregate occurrence counts by structural category, call target where available, and enclosing loop depth. Also record maximum loop depth, the presence of direct recursion and cache patterns, and repeated-expression multiplicities.
3. Compare the two profiles by category and retain entries with different counts, depths, or presence indicators. Preserve the student and reference values separately together with the source intervals supporting each difference.
4. Serialize these entries as category, student observation, reference observation, and associated source locations. Include this contrast with the measured execution profile in the teacher context $c = (p, a)$.

Token-level AST weighting. The second AST-derived signal is the token-level weighting map used in the student loss. Following the notation in Section 3, let G_y denote the set of efficiency-relevant AST regions selected from the student rollout y. We construct G_y by selecting nodes in A_y that correspond to efficiency-related structures, including loops, repeated function calls, recursion, caching, data structure usage, and repeated computation. Region selection uses the student AST alone and is independent of the structural contrast with the reference program. For loops and branches, select the target, iterator, and condition expressions that are present in the corresponding node. For calls, container operations, indexed updates, and repeated expressions, select the corresponding expression or update spans. For detected cache patterns, include the guard, indexed return, and indexed write spans, and include cache decorators when present. Function and loop bodies are not selected in their entirety solely because the enclosing node is recursive or contains a loop. Nested constructs are visited individually, and overlapping source regions are merged before token alignment.

Alignment uses the exact student source text and the tokenizer offset mapping for the generated response. AST line and column positions are converted to absolute source offsets, with byte-based parser positions converted to the tokenizer's coordinate system where necessary. If executable code is extracted from a response containing delimiters, the retained substring offset is included when mapping source positions back to response tokens. Let $[s_i, e_i)$ denote the source-aligned interval of token y_i and $[a, b)$ an interval in G_y. The overlap rule is

$$\alpha_i = \gamma \quad \Longleftrightarrow \quad \exists [a, b) \in G_y \text{ such that } \max(s_i, a) < \min(e_i, b).$$

Tokens without a nonempty source interval retain the default weight. Tokens overlapping with these regions receive larger weights in the OPD loss, while tokens outside these regions keep the default weight. Formally, for token position i, we assign

$$\alpha_i = \begin{cases} \gamma, & \text{if token } y_i \text{ overlaps with a region in } G_y, \\ 1, & \text{otherwise,} \end{cases}$$

where $\gamma > 1$ controls the strength of AST-guided upweighting. The resulting weights are applied after the student rollout has been completed, executed, and analyzed. They do not affect the student sampling process itself, and they are separate from the AST structural profile a provided to the teacher.

Scope and interpretation. The AST rules encode structural relevance rather than measured causal attribution to individual tokens. Indirect recursion, dynamically resolved calls, and semantic equivalence between different algorithms are outside these syntactic rules. Structural comparison operates

on category-level profiles, while token alignment is performed entirely within the student response, so corresponding token positions in the reference program are not required. Formatting changes preserve the parsed structure, with offsets recomputed for the resulting source text, but identifier renaming can change named-call and repeated-expression matches. The existing ablation evaluates the contribution of AST-guided weighting within EffiCode-OPD.

H Ability of Teacher Model

We report the direct code-generation performance of the largest model in each model family used as the teacher in EffiCode-OPD. This evaluation directly measures the quality of teacher-generated programs under the same benchmark protocol as the main experiments. It is separate from OPD training: the models in Table 7 generate solutions directly, without student distillation. For this direct-generation evaluation, each teacher model receives only the standard few-shot code-generation prompt, consisting of several example programming problems and solutions followed by an instruction to generate a correct and efficient program with low execution time and low algorithmic complexity. No privileged execution profile, reference code, AST contrast, or student rollout is provided in this teacher-generation setting.

In EffiCode-OPD training, when reference code is included in the teacher-side privileged context, the prompt is constrained to use it only as efficiency context rather than as text to be copied. Specifically, the teacher is conditioned on the problem, the student-generated prefix, the reference program, execution measurements, and AST-level differences, and is prompted to continue the student prefix toward a correct and efficient program. The next-token likelihoods used by OPD are read from the teacher logits under this continuation context; they are not requested as textual output. The prompt does not ask the teacher to restart from the reference solution, output the reference solution, or continue from the reference code. This design keeps the reference program as privileged information for conditioning teacher-side continuation, while the OPD loss is still computed on the student rollout tokens.

Teacher Model	EffiBench					EFFIBENCH-X				
	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
Code Llama 34B	0.91	0.78	0.66	0.68	28.7	0.92	1.03	0.77	0.96	20.3
Qwen3 32B	**0.65**	**0.70**	**0.58**	0.71	**34.8**	**0.92**	**0.67**	**0.68**	**0.88**	**31.2**
Gemma 3 27B	0.79	0.81	0.69	0.82	29.6	1.04	0.83	0.81	1.01	24.7
Qwen2.5-Coder 32B	0.71	0.74	0.61	**0.69**	33.2	0.96	0.71	0.72	0.91	29.5

Table 7: Direct code-generation performance of the largest teacher model in each model family. The metrics follow the same definitions as the main experiments. Lower is better for ET, NET, MU, and NMU, while higher is better for Pass@1.

Table 7 shows that the largest teacher models provide strong direct generation performance across both benchmarks. Qwen3 32B achieves the best Pass@1 on both EffiBench and EFFIBENCH-X, and also obtains the best ET, NET, and MU on EffiBench. Qwen2.5-Coder 32B is close to Qwen3 32B on EffiBench and gives the best NMU on that benchmark. On EFFIBENCH-X, Qwen3 32B gives the best value across all reported metrics in this table. These results support the use of large models as teachers, since they provide strong direct code-generation performance before being used for token-level supervision.

I Sensitivity Analysis of γ

The hyperparameter γ controls how strongly efficiency-relevant AST regions are upweighted in the student loss. We train Code-Llama-7B with different values of γ and evaluate the resulting models on EffiBench and EffiBench-X. As shown in Table 8, increasing γ from 1.2 to 2.0 reduces NET and NMU and increases Pass@1 on both benchmarks. Among the evaluated settings, $\gamma = 2.0$ achieves the best values for all three reported metrics on both benchmarks and is used as the default setting.

γ	EffiBench			EffiBench-X		
	NET ↓	NMU ↓	Pass@1 ↑	NET ↓	NMU ↓	Pass@1 ↑
1.2	1.01	0.87	20.8	1.25	1.16	13.2
1.4	0.98	0.82	21.3	1.21	1.12	13.6
1.6	0.94	0.78	21.8	1.17	1.08	14.0
1.8	0.91	0.74	22.2	1.14	1.05	14.3
2.0	**0.89**	**0.72**	**22.5**	**1.12**	**1.03**	**14.5**
1.5	0.96	0.80	21.6	1.19	1.10	13.8
3.0	0.95	0.78	21.4	1.18	1.09	13.6

Table 8: Sensitivity analysis of γ for AST-guided token weighting using Code-Llama-7B on EffiBench and EffiBench-X. Lower is better for NET and NMU, while higher is better for Pass@1.

J Trajectory Analysis

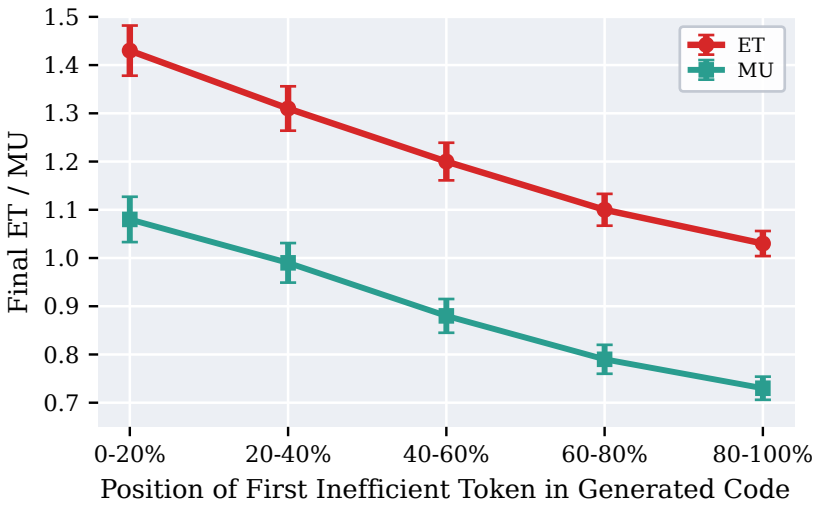


Figure 5: Trajectory analysis on Code-Llama-7B and EffiBench. The x-axis groups generated programs by where the first inefficient token appears in the generated sequence. For example, 0–20% means that the first inefficient token appears within the first 20% of the generated code. The y-axis reports the average ET and MU of the complete generated programs after execution.

We further analyze how early efficiency-critical deviations affect the final efficiency of generated programs. Code generation is path-dependent because early tokens often determine algorithmic plans, data structures, loop structures, and memory access patterns. Once a generation trajectory enters an inefficient implementation, later tokens tend to preserve and complete that structure. As shown in Figure 5, programs whose first efficiency-critical decision appears earlier have higher final normalized ET and MU. This trend indicates that efficiency errors accumulate along the generation trajectory rather than appearing only as local token-level mistakes. It also explains why program-level feedback is insufficient for efficient code generation. Therefore, applying token-level supervision to efficiency-relevant spans allows EffiCode-OPD to correct inefficient decisions early and leads to more efficient generated programs.

K Effect of Reference Code

To examine the robustness of EffiCode-OPD to reference quality, we compare different sources of reference code while keeping the Code-Llama-7B student model, teacher model, rollout procedure, and training objective fixed.

For datasets with released reference solutions, we use the provided reference code as the default reference. For datasets without released references, we sample multiple programs from an auxiliary model, retain only candidates that pass all tests, profile their execution time and memory usage, and select the correct candidate with the best measured efficiency. In the sensitivity study, we instantiate this source with ground-truth reference code, 32B-model sampled reference code, and medium-scale-model sampled reference code. For each problem, the selected reference must be functionally correct and more efficient than the student rollout used for efficiency distillation; otherwise, that rollout is excluded from reference-conditioned distillation.

Reference source	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
Ground Truth	**1.03**	**0.89**	**0.73**	**0.72**	**22.5**
Qwen3-32B	1.06	0.92	0.76	0.77	22.4
Qwen3-14B	1.14	1.01	0.85	0.89	22.1

Table 9: Reference-code quality sensitivity for Code-Llama-7B on EffiBench. All variants keep the same OPD training framework and differ only in the source of the reference code included in the teacher-side privileged context.

Table 9 shows that reference quality mainly affects efficiency metrics. The ground-truth reference gives the best ET, NET, MU, and NMU. When the reference is generated by sampled code instead of taken from the benchmark, efficiency degrades with the measured quality of the selected reference: ET increases from 1.06 with Qwen3-32B references to 1.14 with Qwen3-14B references, and NMU increases from 0.77 to 0.89. Pass@1 remains relatively stable across reference conditions, ranging from 22.1 to 22.5.

This suggests that higher-quality reference code leads to stronger efficiency improvements in OPD performance; in the absence of ground-truth solutions, selecting the highest-quality correct reference available within our framework can still provide useful training signal.

K.1 Student-Generated References as a Possible Extension

An extension would obtain reference programs from the student's own rollouts to reduce dependence on externally supplied reference code. For each training problem, the existing $K = 5$ rollouts would be filtered by correctness and profiled under the same execution environment. For a given correct rollout, another correct candidate could serve as its reference when its measured runtime and memory are both no worse and at least one is better. Candidate improvements would need to be checked under repeated profiling to distinguish them from execution noise. When several candidates satisfy this criterion, a fixed preference rule would be needed to resolve runtime and memory trade-offs. Rollouts without an eligible reference would be excluded from this variant's efficiency distillation updates.

The external teacher and AST-guided loss would remain unchanged, with the selected student candidate supplying the reference code, execution measurements, and structural comparison. This extension would change the source of the reference program while retaining teacher supervision. Problems with fewer than two correct candidates or no efficiency-dominating pair would supply no reference-based training signal under this rule. An evaluation would therefore need to report reference availability, retained training samples, profiling costs, correctness, and execution efficiency.

L Alignment Across Different Tokenizers

For student and teacher models with different tokenizers, we perform distillation over shared code continuation strings. For example, the identifier `total_count` can contain different numbers of tokens under the two tokenizers while representing the same source text. The alignment procedure is used for the corresponding teacher comparisons in Table 3. The main experiments retain the shared-vocabulary token-level objective in Section 3.

Shared source prefixes. For each retained student rollout, we map its generated tokens to source intervals and tokenize the same source text with the teacher tokenizer. Both offset mappings are converted to the same character coordinate system. We use the intersection of their token boundaries as supervision positions, so that the student and teacher condition on identical code prefix text at each position. The student retains its generated prefix tokens, while the teacher uses its own encoding of the same prefix. The problem and privileged efficiency context precede the code prefix in the teacher input, following P2 in Appendix M. The student receives only the problem and its code prefix.

Shared continuation candidates. At each aligned position j, we sample short continuations from both models under their respective conditioning inputs, truncate them at the next newline or the configured length limit, and merge duplicate strings to obtain a shared candidate set $\mathcal{U}_j$. Candidate generation uses the student code-generation instruction and the teacher continuation instruction in Appendix M. Each model encodes every candidate with its own tokenizer while keeping the prefix token sequence fixed. We verify that appending the candidate tokens decodes to the unchanged prefix followed by the candidate text, and retain only candidates satisfying this check for both models. Positions with fewer than two distinct valid candidates are omitted.

Continuation distribution matching. Let h_m^j denote the conditioning token sequence for model $m \in \{s, t\}$ at position j, and let $z^{m,u}$ denote its token sequence for candidate u. We compute the candidate score by teacher forcing and normalize it over the shared candidate set

$$s_m^j(u) = \sum_{k=1}^{|z^{m,u}|} \log \pi_m\big(z_k^{m,u} \mid h_m^j, z_{<k}^{m,u}\big), \qquad q_m^j(u) = \frac{\exp s_m^j(u)}{\sum_{v \in \mathcal{U}_j} \exp s_m^j(v)}.$$

Scores use the sum of token log probabilities without division by token count, since the tokenization length of the same string can differ between models. Each score refers to the specified encoding path for that model and does not marginalize over alternative token sequences that decode to the same string. The resulting objective is a surrogate KL over the finite candidate set, distinct from the full-vocabulary token-level KL used with a shared tokenizer. Candidate strings and teacher scores are held fixed during each update, and gradients flow through the student scores.

AST weighting in source coordinates. For each retained position j, let I_j denote the interval in the original student source from that position to the next newline or the length limit used for continuation extraction. Let ρ_j be the fraction of characters in I_j covered by the union of efficiency-relevant AST regions G_y. We assign a weight and compute the rollout loss as

$$\rho_j = \frac{|I_j \cap \bigcup G_y|}{|I_j|}, \qquad w_j = 1 + (\gamma - 1)\rho_j, \qquad \mathcal{L}_{\text{cross}}(x, y, c) = \frac{\sum_j w_j D_{\text{KL}}(q_s^j \,\|\, q_t^j)}{\sum_j w_j},$$

where interval sizes count source characters and the sums include retained positions with nonempty source intervals. We average this loss over eligible correct student rollouts, excluding rollouts without a retained supervision position.

Interpretation of teacher comparisons. The comparisons across model families include both a change of teacher and, when tokenizers differ, a change from token-level distribution matching to candidate-level distribution matching. They therefore measure the performance of the evaluated teacher and alignment configurations jointly and do not isolate model family from the choice of distillation objective.

M PROMPT

Figure 6 provides the prompt templates used in EffiCode-OPD. These prompts cover all LLM calls in our framework. P1 is the prompt used by the student model during rollout collection and final inference. P2 is used during training, after a student rollout has been completed, executed, profiled, and paired with an efficient reference program. P3 is used when a dataset does not provide reference code; generated candidates are filtered by correctness tests and execution efficiency before being used as references. P4 is used to evaluate the direct generation ability of teacher models and is not part of the student inference pipeline. AST parsing, runtime profiling, memory measurement, and

P1. Standard code-generation prompt for student rollout and inference.

```
You are given a programming problem.  Write a correct and efficient program that solves
the problem.  Return only executable code without explanation.
Problem:
[PROBLEM_STATEMENT]
```

P2. Teacher privileged continuation prompt for OPD training.

```
You are given a programming problem, a student-generated program prefix, and privileged
efficiency context from a more efficient reference program.  Continue the student prefix
into a correct and efficient program.  Use the reference code, execution profile, and AST
structural contrast only as guidance for improving efficiency.  Return only executable
code without explanation.
Problem:
[PROBLEM_STATEMENT]
Efficient reference code:
[REFERENCE_CODE]
Execution profile:
Student runtime:  [STUDENT_RUNTIME]; student memory:  [STUDENT_MEMORY].
Reference runtime:  [REFERENCE_RUNTIME]; reference memory:  [REFERENCE_MEMORY].
AST structural contrast:
[AST_STRUCTURAL_CONTRAST]
Student prefix:
[STUDENT_PREFIX]
```

P3. Reference candidate generation prompt.

```
You are given a programming problem.  Generate a correct and efficient solution with low
execution time and low memory usage.  Return only executable code without explanation.
Problem:
[PROBLEM_STATEMENT]
```

P4. Direct teacher generation prompt.

```
You are given example programming problems and solutions, followed by a new programming
problem.  Generate a correct and efficient program for the new problem.  Return only
executable code without explanation.
Examples:
Example 1 problem:
[EXAMPLE_PROBLEM_1]
Example 1 solution:
[EXAMPLE_SOLUTION_1]
Example 2 problem:
[EXAMPLE_PROBLEM_2]
Example 2 solution:
[EXAMPLE_SOLUTION_2]
Example 3 problem:
[EXAMPLE_PROBLEM_3]
Example 3 solution:
[EXAMPLE_SOLUTION_3]
Problem:
[PROBLEM_STATEMENT]
```

Figure 6: Prompt templates used in EffiCode-OPD.

AST-guided token weighting are implemented by program analysis and execution tools, so they do not require additional LLM prompts.

Teacher input serialization and token alignment. In P2, the problem, reference code, execution profile, and AST structural contrast precede the student prefix. Let $q_t(x, c)$ denote the token sequence containing these prompt fields and the continuation boundary immediately before `[STUDENT_PREFIX]`. The teacher distribution at position i is evaluated as

$$p_t^i = \pi_t(\cdot \mid q_t(x, c) \parallel y_{<i}), \tag{1}$$

where $\parallel$ denotes token sequence concatenation. The conditioning notation $\pi_t(\cdot \mid x, y_{<i}, c)$ in the main text specifies the available information, while this expression specifies its serialized order. For a chat template, $q_t(x, c)$ includes the formatted instruction and context messages followed by the assistant opening markers, and $y_{<i}$ forms the unfinished assistant response. The student prefix is appended as rollout tokens, with no additional whitespace, delimiters, end-of-turn markers, or instructions after it. With one-based token positions and $m = |q_t(x, c)|$, the teacher logits at position $m + i - 1$ predict y_i under a causal attention mask. For $i = 1$, these logits are read at the final token of $q_t(x, c)$. Only positions predicting student rollout tokens contribute to the distillation loss.

N Additional Main Results

Model	Method	EffiBench					EFFIBENCH-X				
		ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
Code-Llama-13B	Few-shot	1.64	1.40	1.14	1.52	14.6	1.78	1.62	1.27	1.57	11.5
	SFT	1.60	1.37	1.11	1.48	17.2	1.75	1.61	1.24	1.55	12.8
	PPO	1.36	1.20	0.94	1.30	22.1	1.48	1.42	1.08	1.34	16.2
	EffiLearner	1.49	1.22	1.06	1.36	16.8	1.62	1.55	1.22	1.45	13.6
	PerfCodeGen	1.42	1.10	0.98	1.19	21.6	1.54	1.39	1.17	1.39	14.9
	LLM-as-Critique	1.06	0.94	0.84	0.92	22.4	1.15	1.17	0.96	1.11	16.4
	EffiCode-OPD (Ours)	**0.90**	**0.82**	**0.68**	**0.71**	**25.3**	**0.96**	**1.02**	**0.78**	**0.96**	**18.2**
	Gain(Δ)	↓15.1%	↓12.8%	↓19.0%	↓22.8%	↑12.9%	↓16.5%	↓12.8%	↓18.8%	↓13.5%	↑11.0%
Gemma-3-12B	Few-shot	1.58	1.39	1.20	1.24	17.4	1.80	1.66	1.35	1.60	18.1
	SFT	1.55	1.36	1.17	1.21	20.2	1.77	1.63	1.32	1.57	19.4
	PPO	1.37	1.18	1.05	1.02	23.5	1.59	1.45	1.19	1.33	22.6
	EffiLearner	1.47	1.28	1.12	1.11	21.1	1.68	1.53	1.27	1.43	20.9
	PerfCodeGen	1.40	1.19	1.02	0.97	23.1	1.61	1.42	1.15	1.30	22.1
	LLM-as-Critique	1.00	0.92	0.86	0.90	24.0	1.29	1.16	1.02	1.12	22.9
	EffiCode-OPD (Ours)	**0.84**	**0.80**	**0.69**	**0.74**	**27.1**	**1.08**	**1.01**	**0.86**	**0.99**	**25.2**
	Gain(Δ)	↓16.0%	↓13.0%	↓19.8%	↓17.8%	↑12.9%	↓16.3%	↓12.9%	↓15.7%	↓11.6%	↑10.0%

Table 10: Additional main results on the remaining student models for EffiBench and EffiBench-X. The table follows the same format and evaluation protocol as Table 1. Lower is better for ET, NET, MU, and NMU, while higher is better for Pass@1.

We report additional results for the remaining student models on EffiBench, EFFIBENCH-X, Mercury and ENAMEL. The evaluation settings, baselines, metrics, and teacher selection follow the main experiments.

As shown in Tables 10 and 11, EffiCode-OPD achieves the best results across all reported metrics and model settings.

Model	Method	Mercury					ENAMEL				
		ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑	ET ↓	NET ↓	MU ↓	NMU ↓	Pass@1 ↑
Code-Llama-13B	Few-shot	2.05	1.91	1.48	1.82	9.4	2.18	2.04	1.56	1.95	8.6
	SFT	2.02	1.88	1.45	1.79	10.6	2.14	2.01	1.52	1.92	9.4
	PPO	1.74	1.62	1.29	1.55	13.8	1.87	1.75	1.39	1.68	12.5
	EffiLearner	1.91	1.78	1.42	1.70	11.2	2.04	1.92	1.50	1.82	10.1
	PerfCodeGen	1.80	1.58	1.34	1.51	12.9	1.94	1.72	1.45	1.65	11.6
	LLM-as-Critique	1.39	1.41	1.16	1.36	14.2	1.54	1.56	1.27	1.49	13.0
	EffiCode-OPD (Ours)	**1.18**	**1.23**	**0.94**	**1.15**	**16.0**	**1.30**	**1.36**	**1.03**	**1.25**	**15.2**
	Gain(Δ)	↓15.1%	↓12.8%	↓19.0%	↓15.4%	↑12.7%	↓15.6%	↓12.8%	↓18.9%	↓16.1%	↑16.9%
Gemma-3-12B	Few-shot	1.95	1.86	1.47	1.78	14.6	2.08	1.99	1.55	1.91	13.5
	SFT	1.91	1.82	1.44	1.74	15.8	2.03	1.95	1.52	1.88	14.6
	PPO	1.70	1.59	1.31	1.52	18.9	1.84	1.72	1.41	1.65	17.4
	EffiLearner	1.83	1.70	1.39	1.63	16.7	1.96	1.84	1.49	1.76	15.5
	PerfCodeGen	1.74	1.57	1.27	1.47	18.2	1.88	1.70	1.38	1.60	16.8
	LLM-as-Critique	1.32	1.33	1.14	1.29	19.1	1.47	1.48	1.25	1.42	17.9
	EffiCode-OPD (Ours)	**1.10**	**1.16**	**0.92**	**1.08**	**21.5**	**1.22**	**1.28**	**1.01**	**1.19**	**20.1**
	Gain(Δ)	↓16.7%	↓12.8%	↓19.3%	↓16.3%	↑12.6%	↓17.0%	↓13.5%	↓19.2%	↓16.2%	↑12.3%
Code-Llama-7B	Few-shot	2.28	2.05	1.52	2.03	7.1	2.42	2.18	1.60	2.15	6.4
	SFT	2.24	2.01	1.49	2.00	8.2	2.38	2.15	1.57	2.11	7.0
	PPO	1.95	1.76	1.33	1.70	12.4	2.10	1.91	1.45	1.84	10.2
	EffiLearner	2.12	1.92	1.50	1.86	8.8	2.27	2.09	1.58	2.00	7.8
	PerfCodeGen	2.03	1.70	1.42	1.62	11.7	2.18	1.87	1.54	1.78	8.9
	LLM-as-Critique	1.50	1.45	1.22	1.37	12.1	1.64	1.57	1.34	1.51	9.8
	EffiCode-OPD (Ours)	**1.27**	**1.26**	**0.99**	**1.08**	**13.9**	**1.39**	**1.38**	**1.10**	**1.20**	**11.5**
	Gain(Δ)	↓15.3%	↓13.1%	↓18.9%	↓21.2%	↑14.9%	↓15.2%	↓12.1%	↓17.9%	↓20.5%	↑17.3%
Qwen3-8B	Few-shot	2.18	2.00	1.60	1.68	9.3	2.36	2.14	1.70	1.82	8.7
	SFT	2.15	1.96	1.62	1.63	11.0	2.39	2.13	1.72	1.84	10.4
	PPO	1.88	1.63	1.48	1.39	14.0	2.11	1.88	1.55	1.61	13.6
	EffiLearner	2.04	1.82	1.57	1.54	12.3	2.18	2.04	1.79	1.74	12.9
	PerfCodeGen	1.93	1.69	1.45	1.44	14.1	2.05	1.95	1.63	1.52	13.2
	LLM-as-Critique	1.43	1.39	1.14	1.25	14.2	1.54	1.43	1.22	1.33	13.8
	EffiCode-OPD (Ours)	**1.21**	**1.24**	**0.94**	**1.12**	**15.6**	**1.36**	**1.28**	**1.05**	**1.19**	**15.4**
	Gain(Δ)	↓15.4%	↓10.8%	↓17.5%	↓10.4%	↑9.9%	↓11.7%	↓10.5%	↓13.9%	↓10.5%	↑11.6%
Qwen3-14B	Few-shot	1.96	1.74	1.49	1.47	15.1	2.12	1.91	1.54	1.75	15.8
	SFT	1.92	1.70	1.51	1.48	17.6	2.08	1.87	1.56	1.72	16.5
	PPO	1.70	1.53	1.38	1.36	20.9	2.02	1.77	1.43	1.49	19.1
	EffiLearner	1.82	1.64	1.36	1.37	17.0	2.10	1.88	1.48	1.57	16.8
	PerfCodeGen	1.75	1.54	1.28	1.29	21.2	1.92	1.65	1.39	1.42	19.9
	LLM-as-Critique	1.15	1.09	1.03	1.13	21.0	1.61	1.14	1.18	1.31	19.7
	EffiCode-OPD (Ours)	**0.97**	**1.00**	**0.84**	**1.04**	**23.7**	**1.22**	**1.02**	**0.96**	**1.16**	**21.7**
	Gain(Δ)	↓15.7%	↓8.3%	↓18.4%	↓8.0%	↑11.8%	↓24.2%	↓10.5%	↓18.6%	↓11.5%	↑9.0%
Qwen2.5-Coder-7B	Few-shot	2.10	1.96	1.58	1.67	12.2	2.30	2.12	1.72	1.89	13.9
	SFT	2.12	1.93	1.59	1.69	13.1	2.32	2.10	1.74	1.91	15.0
	PPO	1.93	1.64	1.44	1.45	16.4	2.19	1.86	1.41	1.56	18.0
	EffiLearner	2.08	1.82	1.53	1.60	14.7	2.18	1.97	1.63	1.76	17.4
	PerfCodeGen	2.04	1.76	1.50	1.55	15.3	2.24	1.95	1.56	1.66	17.2
	LLM-as-Critique	1.73	1.56	1.34	1.42	16.0	2.03	1.75	1.30	1.39	19.1
	EffiCode-OPD (Ours)	**1.48**	**1.36**	**1.14**	**1.25**	**17.6**	**1.72**	**1.49**	**1.18**	**1.27**	**20.5**
	Gain(Δ)	↓14.5%	↓12.8%	↓14.9%	↓12.0%	↑7.3%	↓15.3%	↓14.9%	↓9.2%	↓8.6%	↑7.3%
Qwen2.5-Coder-14B	Few-shot	1.92	1.80	1.55	1.61	18.2	2.13	2.01	1.78	1.82	18.3
	SFT	1.88	1.82	1.53	1.62	20.1	2.09	2.03	1.74	1.81	19.5
	PPO	1.58	1.44	1.40	1.38	23.7	1.79	1.70	1.58	1.49	22.1
	EffiLearner	1.77	1.66	1.43	1.47	21.0	2.01	1.84	1.49	1.66	20.9
	PerfCodeGen	1.75	1.63	1.45	1.45	21.5	1.92	1.88	1.42	1.61	21.4
	LLM-as-Critique	1.44	1.42	1.18	1.29	22.9	1.74	1.51	1.51	1.41	22.0
	EffiCode-OPD (Ours)	**1.36**	**1.30**	**1.05**	**1.14**	**25.1**	**1.50**	**1.40**	**1.32**	**1.30**	**24.0**
	Gain(Δ)	↓5.6%	↓8.5%	↓11.0%	↓11.6%	↑5.9%	↓13.8%	↓7.3%	↓7.0%	↓7.8%	↑8.6%

Table 11: Additional main results on Mercury and ENAMEL. The table includes the remaining student models and all student models from Table 1, and follows the same format and evaluation protocol. Lower is better for ET, NET, MU, and NMU, while higher is better for Pass@1.