LLM4EFFI: Leveraging Large Language Models to Enhance Code Efficiency and Correctness

Anonymous ACL submission

Abstract

Large Language Models have demonstrated impressive capabilities in generating syntactically and functionally correct code. However, most existing research has primarily focused on the correctness of generated code, while efficiency remains relatively underexplored. Recent efforts have attempted to enhance efficiency by refining the initially generated code. Nonetheless, such post hoc optimizations are inherently constrained by the original algorithmic design and overall logic, often yielding only marginal gains. In this work, we propose LLM4EFFI, a novel framework that enables LLMs to generate code that balances both efficiency and correctness. LLM4EFFI decomposes the efficiency optimization process into two distinct stages: algorithmic exploration at the logical level and implementation optimization at the code level. Correctness is subsequently ensured through an adaptive testing process based on synthetic test cases. By prioritizing efficiency early in the generation process and refining for correctness afterward, LLM4EFFI introduces a new paradigm for efficient code generation. Experimental results show that LLM4EFFI consistently improves both efficiency and correctness of generated code, achieving state-of-the-art performance on three code efficiency benchmarks across five diverse LLM backbones.

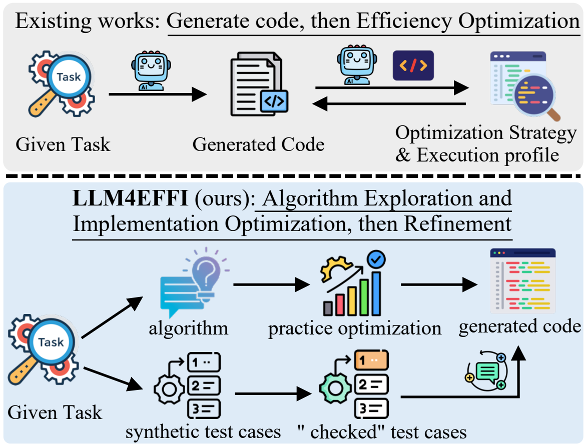

Figure 1: Comparison of LLM4EFFI with existing methods. Existing approaches typically generate code first and then attempt to improve efficiency through strategy or execution profiling. In contrast, LLM4EFFI reverses this order by prioritizing efficiency during generation, followed by refinement to ensure correctness.

1 Introduction

Large Language Models (LLMs), particularly Code LLMs, are transforming the field of software engineering at an unprecedented pace. A significant area of advancement lies in automated code generation (Liu et al., 2023; Li et al., 2024a), where LLMs such as GPT-4o (OpenAI, 2024), Gemini (Team et al., 2023), the DeepSeek series (DeepSeek-AI, 2024a), and the Qwen series (Yang et al., 2024a,b) have demonstrated remarkable capabilities. These LLMs have consistently broken new ground on code completion and generation benchmarks, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), LiveCodeBench (Jain et al., 2024), and BigCodeBench (Zhuo et al., 2025).

While LLMs have achieved impressive accuracy in code generation, practical software engineering requires more than correctness—it also demands efficiency (Shi et al., 2024; Niu et al., 2024). In real-world scenarios, even functionally correct code often needs manual optimization before deployment, which undermines the promise of truly "out-of-the-box" code generation. Therefore, generating code that is both correct and efficient is essential; yet automating this process remains largely unexplored.

Recent preliminary works (Huang et al., 2024b; Waghjale et al., 2024) have explored feedback-based approaches to improving the execution efficiency of generated code. As shown in Fig. 1, these methods follow a "generate-then-optimize" paradigm, where the model first generates code and subsequently refines it through strategy prompts or execution profiling. However, this paradigm raises a fundamental question:

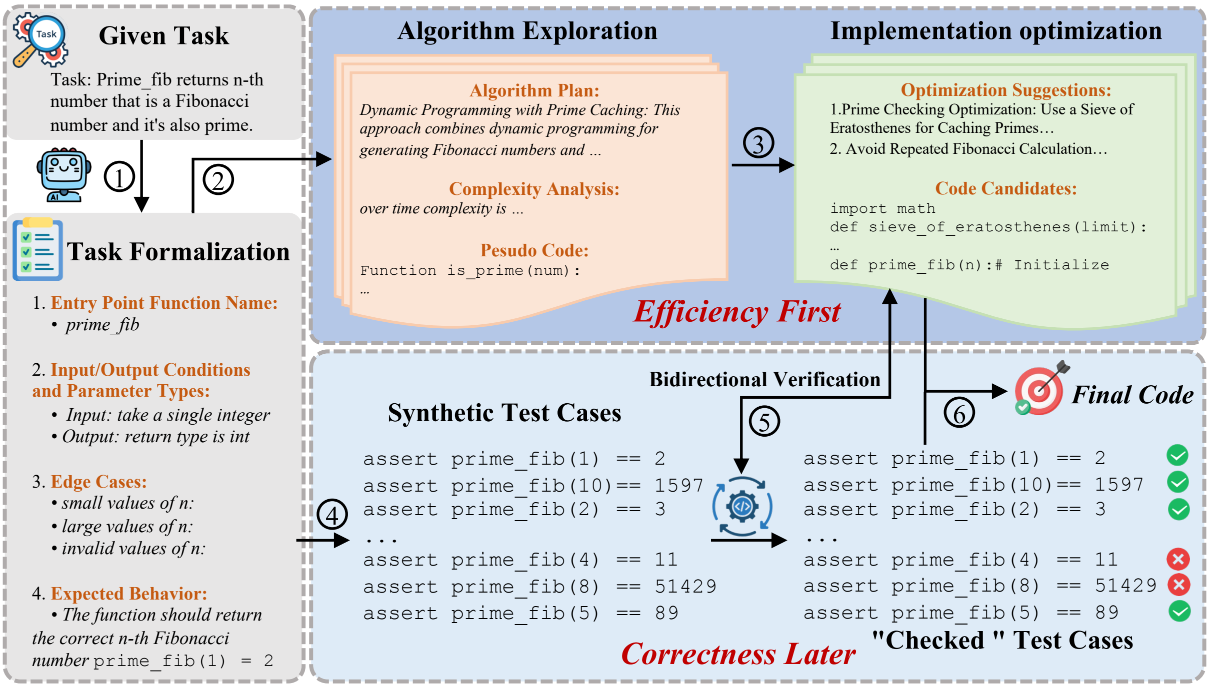

Figure 2: The workflow of LLM4EFFI. Given a programming task, LLM4EFFI formalizes it into a code-oriented description, generates optimal algorithms and pseudocode in logic domain, and then produces implementation suggestions in code domain. LLM4EFFI synthesizes test cases and uses a verification-based adaptive framework to evaluate candidate solutions. The final code is selected based on the highest pass rate of the "checked" test cases.

> *"Is generating code first and then optimizing for efficiency truly the optimal solution for efficient code generation?"*

To investigate this, we conduct a detailed analysis of concrete examples, with representative cases provided in Fig. 22 and 23 in Appendix B. This analysis reveals that the "generate-then-optimize" paradigm is fundamentally limited by the algorithmic design and structural choices made during initial code generation, often yielding only incremental efficiency gains. Such constraints hinder deeper, algorithm-level restructuring necessary for achieving substantial improvements. In contrast, when human developers write high-quality code, whether in practical software engineering or algorithmic education, they typically begin by exploring multiple potential solutions at a logical level. They analyze the task's constraints, compare algorithmic alternatives and their computational complexities, and then proceed to implementation. During this stage, they apply appropriate coding techniques to optimize performance, followed by debugging and refinement to ensure correctness and quality.

Inspired by this insight, we propose LLM4EFFI (Large Language Model for Code Efficiency), a novel paradigm that enables LLMs to generate code that is both efficient and correct, as illustrated in Fig. 2. Given a programming task described in natural language, LLM4EFFI first formalizes it into a code-oriented problem specification. That is, it transforms the broad task description into a clear and well-defined coding problem, enabling accurate interpretation by the LLM. LLM4EFFI then guides the LLM through logic-level reasoning and algorithmic exploration. It considers multiple candidate algorithms, analyzes their computational complexities, and generates corresponding pseudocode. Based on these designs, LLM4EFFI derives implementation strategies and proceeds to code generation, incorporating optimization at the implementation level. This reflects the understanding that high-quality code requires not only sound algorithmic foundations but also attention to practical implementation. To ensure functional correctness while targeting efficiency, LLM4EFFI introduces a bidirectional verification-based adaptive testing framework that dynamically constructs and validates synthetic test cases. The generated code is iteratively executed and refined based on these "checked" test cases. Ultimately, the solution with the highest pass rate across the "checked" test cases is selected as the final output.

LLM4EFFI introduces two key uniqueness: Uniqueness 1: *Separation of Efficiency Optimiza-*

tion into Logic and Code Domains. LLM4EFFI decomposes efficiency optimization into two distinct stages: the logic domain and the code domain. The logic domain focuses on identifying optimal algorithmic strategies, while the code domain targets low-level implementation refinements. This separation breaks down the complex task of code efficiency optimization into manageable and complementary phases, making the overall process more systematic, interpretable and targeted.

Uniqueness 2: ***Reordering the Optimization of Correctness and Efficiency.*** The order of optimizing efficiency and correctness plays a crucial role. LLM4EFFI prioritizes efficiency first, enabling broader algorithmic exploration and allowing multiple efficient solutions to emerge. Correctness is then incrementally ensured across these candidates. This strategy avoids prematurely constraining the solution space by enforcing correctness too early, which limit optimization potential. By reversing traditional order, LLM4EFFI unlocks greater potential for substantial efficiency gains.

We evaluate LLM4EFFI on three recently proposed code efficiency benchmarks: EvalPerf (Liu et al., 2024), ENAMEL (Qiu et al., 2024), and Mercury (Du et al., 2024). Experimental results demonstrate that LLM4EFFI consistently improves both code correctness and efficiency across diverse LLM backbones, achieving state-of-the-art performance on efficiency-related metrics. Overall, the contributions are threefold; code available at link[1]:

- We propose LLM4EFFI, the first LLM-based framework that integrates efficiency and correctness as joint optimization objectives.
- We introduce two key innovations: (i) Separation of efficiency optimization into logic and code domains and (ii) Reordering the optimization of correctness and efficiency, which offers a new paradigm for the code efficiency community.
- Extensive experiments and analysis on three benchmarks across five different LLM backbones demonstrate the effectiveness and robustness of LLM4EFFI in efficient code generation.

2 Related Works

2.1 LLMs for Code Generation

LLMs have been widely applied to coding tasks and have demonstrated strong performance across a range of code scenarios. Most existing research focuses on improving code generation quality, with numerous techniques proposed for this purpose. Some methods aim to improve the quality of synthetic code data (Wei et al., 2024; Luo et al., 2024; Lei et al., 2024), improve self-consistency (Le et al., 2024; Huang et al., 2024a), or incorporate feedback from human or LLM annotations (Chen et al., 2024; Wu et al., 2023; Tang et al., 2023). Other approaches utilize multi-agent collaboration frameworks to enhance code generation (Zhong et al., 2024; Shinn et al., 2023; Islam et al., 2024; Madaan et al., 2023; Li et al., 2024b). Despite these advances, most of these methods primarily focus on improving correctness, while largely overlooking the efficiency of the generated code.

2.2 Code Efficiency

Until recently, the efficiency of generated code had received attention from the academic community. Several efficiency-focused benchmarks (HUANG et al., 2024; Du et al., 2024; Liu et al., 2024; Qiu et al., 2024) have been introduced to more comprehensively evaluate the ability of LLMs to produce efficient code. However, empirical evaluations on these benchmarks show that current LLMs still face significant challenges in efficient code generation. To address this, recent work such as ECCO (Waghjale et al., 2024) adopts self-refinement, prompting LLMs to consider possible optimization strategies and iteratively refine their outputs. Effi-Learner (Huang et al., 2024b) proposes a self-optimization framework that leverages execution overhead profiles, feeding them back into the LLM to revise the code and reduce runtime overhead. However, these methods all follow the "generate-then-optimize" paradigm, rather than starting with the goal of generating both efficient and correct code from the beginning.

3 Methodology

Problem Formulation. In the code efficiency task, each instance is defined as a pair (Q, T_h), where Q denotes the task description, and T_h represents the hidden test cases. The goal is to generate a code solution S that passes all the hidden test cases while achieving the highest efficiency (i.e., the shortest execution time). Notably, to better simulate real-world scenarios, we assume that no public test cases are available. T_h is only used during the evaluation stage and is not visible during the efficiency and correctness optimization stages.

[1] https://anonymous.4open.science/r/LLM4EFFI-04B2

3.1 Overview

As shown in Fig.2, given a programming task in natural language, LLM4EFFI first formalizes it into a code-oriented problem specification (**Section §3.2**). LLM4EFFI then prompts the LLM to perform logic-domain reasoning, generating multiple algorithmic candidates and corresponding pseudocode (**Section §3.3**). Based on these, LLM4EFFI derives detailed implementation strategies and generates optimized code (**Section §3.4**). To ensure correctness, it synthesizes test cases and applies a bidirectional verification-based adaptive framework to validate these synthetic test cases. These "checked" test cases are then used to evaluate and refine the candidate code solutions (**Section §3.5**).

3.2 Task Formalization

In the initial task formalization stage, LLM4EFFI ensures that the task description is clear and unambiguous, which is crucial for subsequent stages. As highlighted by Han et al. (2024), errors in LLM-generated code often arise from an insufficient or unclear understanding of the task. Therefore, LLM4EFFI prompts the LLM to interpret the task from four key dimensions: (i) entry point function name, (ii) input/output conditions and parameter types, (iii) edge cases, and (iv) expected behavior. Based on these aspects, the LLM is further guided to engage in self-reflection to confirm whether it has fully understood the task, thus laying a solid foundation for subsequent stages. Formally, this process is represented as : $Q \rightarrow Q_{formal} \overset{\text{check}}{\longleftrightarrow} Q$.

3.3 Algorithmic Exploration in Logic Domain

For the formalized task, LLM4EFFI prompts the LLM to engage in algorithmic reasoning at the logical level, rather than generating code directly. This approach mirrors the workflow of human programmers, who first perform abstract, high-level reasoning before moving to implementation. The LLM is guided to explore multiple optimal algorithms, analyze their complexities, and express the entire reasoning process using pseudocode. Formally, this is denoted as: $Q_{formal} \rightarrow \{Algo, Cplx, Pseudo\}$, where $Algo$ represents the algorithm plan, $Cplx$ denotes the complexity analysis, and $Pseudo$ refers to the corresponding pseudocode.

3.4 Implementation in Code Domain

High-quality code requires not only well-designed algorithms but also careful optimization at the implementation level. Even when the underlying algorithm is the same, different implementation choices can lead to substantial variations in code efficiency (Shypula et al., 2024; Coignion et al., 2024). Based on the algorithmic plan and corresponding pseudocode, LLM4EFFI prompts the LLM to generate practical implementation suggestions derived from $Algo$ and $Pseudo$. For example, replacing a manual binary exponentiation routine with Python's built-in pow function.

LLM4EFFI then generates candidate code implementations based on $Algo$, $Pseudo$, and the derived suggestions, while also identifying additional optimization opportunities. Formally, this process is denoted as:

$$\{Algo, Pseudo\} \rightarrow \{Suggs\}$$

$$\{Algo, Pseudo, Suggs\} \rightarrow \{Code\ Candidates\}$$

3.5 Code Correctness Guarantee

To ensure the functional correctness of the generated code while targeting efficiency, LLM4EFFI introduces a bidirectional verification-based adaptive testing framework. The process unfolds as follows: LLM4EFFI first automatically synthesizes a large number of test cases based on the formalized task description Q_{formal}. These test cases are designed to cover a wide range of edge cases, rigorously testing the robustness and reliability of candidate solutions. However, since the synthesized test cases may contain errors, LLM4EFFI applies a bidirectional verification method to validate them. **Forward Verification:** If all candidate code implementations pass a given test case, it is considered trusted. Otherwise, **Reverse Review:** If any candidate fails a test case, LLM4EFFI performs a Q_{formal}-based review to determine whether the test case aligns with the task intent. This includes a semantic consistency check inspired by Test-Driven Development (TDD) practices (Erdogmus et al., 2010). Test cases that pass this review are retained; otherwise, they are discarded. The retained test cases are then marked as "checked" and used to evaluate the code candidates. Any failures during the evaluation phase trigger further refinement. Ultimately, the refined code candidate that passes the most "checked" test cases is selected as the final output. Formally, this process is denoted as:

$$Q_{formal} \rightarrow \{Synthesized\ Test\ Cases\} \xrightarrow[verification]{bidirectional}$$

$$\{Checked\ Test\ Cases\} \xrightarrow[select]{refine} \{Final\ Output\}$$

LLMs	Methods	EvalPerf		Mercury		ENAMEL	
		DPS_norm	Pass@1	Beyond@1	Pass@1	eff@1	Pass@1
Qwen2.5-Coder -32B-Instruct	Instruct	80.92	85.59	76.97	**94.14**	50.44	85.21
	ECCO	82.16	63.56	73.29	89.06	41.89	71.83
	Effi-Learner	82.45	77.11	77.13	91.41	50.12	81.69
	LLM4EFFI (ours)	**86.20** +5.28	**87.30** +1.71	**78.96** +1.99	93.75 -0.39	**51.26** +0.82	**86.62** +1.41
Qwen2.5-72B -Instruct	Instruct	79.29	88.14	72.50	86.72	49.78	83.80
	ECCO	80.06	64.41	74.10	89.84	41.90	72.53
	Effi-Learner	79.90	81.36	77.10	**91.02**	47.42	76.76
	LLM4EFFI (ours)	**84.00** +4.71	**88.98** +0.84	**77.45** +4.95	90.63 +3.91	**51.49** +1.71	**87.32** +3.52
GPT-4o-mini	Instruct	80.04	85.59	69.59	82.81	48.26	80.28
	ECCO	75.18	44.07	72.29	86.33	30.75	57.75
	Effi-Learner	79.80	81.36	73.45	88.67	45.69	77.46
	LLM4EFFI (ours)	**83.78** +3.74	**88.14** +2.55	**74.94** +5.35	**89.45** +6.64	**49.89** +1.63	**80.99** +0.71
GPT-4o	Instruct	79.59	86.70	73.14	87.50	47.63	80.99
	ECCO	80.65	61.02	77.70	92.18	38.63	64.79
	Effi-Learner	79.39	79.67	**79.24**	93.36	48.52	81.69
	LLM4EFFI (ours)	**86.39** +6.80	**88.98** +2.28	77.81 +4.67	**93.75** +6.25	**55.26** +7.63	**83.80** +2.81
DeepSeek-V3	Instruct	80.45	89.84	79.90	94.53	51.14	86.62
	ECCO	81.08	61.84	63.26	74.61	45.84	75.35
	Effi-Learner	79.00	88.14	78.83	92.58	52.22	83.80
	LLM4EFFI (ours)	**87.08** +6.63	**90.67** +0.83	**82.76** +2.86	**96.09** +1.56	**60.41** +9.27	**89.44** +2.82

Table 1: Main Result. Performance of LLM4EFFI and baseline methods on EvalPerf, Mercury, and ENAMEL benchmarks, using five different LLM backbones. Efficiency is evaluated using each benchmark's specific metric.

4 Experiments

We evaluate LLM4EFFI on three code efficiency benchmarks: EvalPerf (Liu et al., 2024), Mercury (Du et al., 2024), and ENAMEL (Qiu et al., 2024). **EvalPerf** uses differential performance evaluation to assess efficiency across different LLMs and solutions. Its efficiency metric, DPS_norm, is calculated by determining the cumulative ratio of the reference solution that is immediately slower than the new solution, normalized by the total number of solutions. This ensures a fair comparison of code efficiency based on reference solutions with varying performance levels. **Mercury** introduces the Beyond metric to evaluate both correctness and code efficiency. The Beyond metric is calculated by normalizing the runtime percentiles of LLM solution samples over the runtime distribution for each task, ensuring consistent runtime comparisons across different environments and hardware configurations. **ENAMEL** evaluates efficiency using the eff@1 metric, which captures the worst-case execution time of a generated code sample across test cases with varying difficulty. The score is then adjusted using a weighted average across these levels to account for hardware fluctuations. The eff@1 value ranges from 0 to 1, with higher values indicating better efficiency; values above 1 indicate performance exceeding expert-level solutions.

4.1 Compared Methods.

We evaluate the direct instruction to generate both correct and efficient code as the **Instruct** baseline. We compare LLM4EFFI against two recent methods for code efficiency: **ECCO** (Waghjale et al., 2024) and **Effi-Learner** (Huang et al., 2024b).

- **ECCO:** A self-refinement method with natural language feedback. It prompts the LLM to generate an initial solution, then asks whether the code can be improved for efficiency, and refines the solution based on the suggested optimizations.
- **Effi-Learner:** First, it generates code using the same instruction prompt as the **Instruct** baseline. It then executes the code on test cases to collect performance profiles, including runtime and memory usage. These profiles, along with the original code, are fed back into the LLM to guide efficiency-oriented refinement. It is worth noting that Effi-Learner relies on test case oracles; in this study, we use the visible test cases provided with each task. In contrast, LLM4EFFI does not depend on any test case oracles—all test cases are synthetically generated within LLM4EFFI itself.

4.2 Experiment Setup.

To comprehensively evaluate the applicability of LLM4EFFI, we selected five different LLM backbones: two proprietary LLMs: GPT-4o (OpenAI,

Models	Methods	EvalPerf		Mercury		ENAMEL	
		DPS_norm	Pass@1	Beyond@1	Pass@1	eff@1	Pass@1
Qwen2.5-Coder-32B-Instruct	**LLM4EFFI**	**86.20**	**87.30**	**78.96**	**93.75**	**51.26**	**86.62**
	Variant-1	77.21 -8.99	80.51 -6.79	77.89 -1.07	93.34 -0.41	48.57 -2.69	81.69 -4.93
	Variant-2	75.75 -10.45	81.36 -5.94	75.86 -3.10	92.19 -1.56	45.68 -5.58	83.10 -3.52
	Variant-3	81.19 -5.01	72.03 -15.27	72.56 -6.40	85.16 -8.59	47.48 -3.78	77.46 -9.16
DeepSeek-V3	**LLM4EFFI**	**87.08**	**90.67**	**82.76**	**96.09**	**60.41**	**89.44**
	Variant-1	79.72 -7.36	84.75 -5.92	81.58 -1.18	94.53 -1.56	53.23 -7.18	88.03 -1.41
	Variant-2	77.07 -10.01	83.05 -7.62	80.10 -2.66	94.53 -1.56	53.62 -6.79	88.73 -0.71
	Variant-3	82.62 -4.46	82.01 -8.66	79.75 -3.01	92.58 -3.51	54.58 -5.83	81.69 -7.75

Table 2: Ablation Study Results. Results of LLM4EFFI, Variant-1, Variant-2, and Variant-3 are presented using Qwen2.5-Coder-32B-Instruct and DeepSeek-V3 as backbones on EvalPerf, Mercury, and ENAMEL benchmarks.

2024) and GPT-4o-mini, and three open-source LLMs, including DeepSeek-V3 (DeepSeek-AI, 2024b), Qwen2.5-72B-Instruct (Yang et al., 2024b), and Qwen2.5-Coder-32B-Instruct (Hui et al., 2024). During the LLM4EFFI process, we set the number of algorithm plans to 5 and the number of synthetic test cases to 20, followed by one iteration to refine the code for correctness. All prompts used for LLM4EFFI and the baselines are provided in Appendix A. For fair comparison, all experiments are conducted with a temperature setting of 0 and repeated three times, with average results reported.

4.3 Main Results.

We compare LLM4EFFI with other methods on the EvalPerf, Mercury, and ENAMEL benchmarks, with results shown in Tab. 1. Direct instruction prompts (Instruct) yield strong performance, suggesting that LLMs possess a basic understanding of correct and efficient code. ECCO provides slight efficiency gains on EvalPerf and Mercury, but these often come at the cost of correctness. On more complex benchmarks like ENAMEL, ECCO shows a drop in both efficiency and correctness, indicating that relying solely on code domain optimization is insufficient. Lacking alignment with the broader logic requirement of the task, such methods often fail to produce effective improvements.

Effi-Learner shows moderate gains in certain settings, such as GPT-4o on Mercury, but its performance is inconsistent across LLMs and benchmarks, often falling short of the Instruct baseline. More importantly, it frequently compromises both efficiency and correctness. This is primarily due to its feedback mechanism, which emphasizes performance metrics (e.g., execution time) while neglecting functional correctness. Lacking algorithmic reasoning, Effi-Learner tends to over-optimize for runtime at the expense of code reliability, ultimately undermining both objectives. In comparison, LLM4EFFI achieves consistent improvements in both correctness and efficiency. Notably, it is the only method that delivers robust performance gains across diverse benchmarks and LLMs. Under DeepSeek-V3, LLM4EFFI improves DPS_norm by 6.63% on EvalPerf, increases eff@1 by 9.27% on ENAMEL, and boosts Pass@1 by 2.82%.

4.4 Ablation Study.

LLM4EFFI incorporates several distinctive design choices, such as separating efficiency optimization into the logic and code domains. To assess the impact of each part, we conduct following variants:

- **Variant-1:** (Without Algorithmic Exploration in the Logic Domain): This variant skips algorithmic exploration for logic-level efficiency optimization. Instead, the LLM directly generates a fixed number of efficient code candidates based on the formalized task, followed by implementation-level optimization. All other steps are identical to those in LLM4EFFI.
- **Variant-2:** (Without Implementation Optimization in the Code Domain): This variant omits the implementation-level optimization step. All other stages remain the same as in LLM4EFFI.
- **Variant-3:** (Without Code Correctness Refinement): In this setting, after generating the efficiency-optimized code, the LLM directly selects the final output it deems most efficient and correct, without the correctness refinement phase.

We conduct the ablation study using Qwen2.5-Coder-32B-Instruct and DeepSeek-V3 as LLM backbones, with the results shown in Tab. 2. The findings indicate that removing any component degrades both efficiency and correctness. Specifically, excluding Algorithmic Exploration in the Logic

Models	Methods	EvalPerf		Mercury		ENAMEL	
		DPS_norm	Pass@1	Beyond@1	Pass@1	eff@1	Pass@1
Qwen2.5-Coder-32B-Instruct	LLM4Effi	**86.20**	**87.30**	**78.96**	93.75	**51.26**	**86.62**
	w/o Uniqueness-1	80.84 -5.36	79.66 -7.64	76.75 -2.21	**94.14** +0.39	50.95 -0.31	80.98 -5.64
	w/o Uniqueness-2	78.07 -8.13	70.34 -16.96	72.83 -6.13	87.11 -6.64	47.62 -3.64	77.46 -9.16
DeepSeek-V3	LLM4Effi	**87.08**	**90.67**	**82.76**	**96.09**	**60.41**	**89.44**
	w/o Uniqueness-1	80.91 -6.17	85.59 -5.08	81.79 -0.97	95.31 -0.78	54.70 -5.71	85.92 -3.52
	w/o Uniqueness-2	80.42 -6.66	79.66 -11.01	62.90 -19.86	74.61 -21.48	53.92 -6.49	84.50 -4.94

Table 3: Uniqueness Analysis Results. Results of LLM4Effi, **w/o Uniqueness-1**, and **w/o Uniqueness-2** using Qwen2.5-Coder-32B-Instruct and DeepSeek-V3 as backbones on EvalPerf, Mercury, and ENAMEL benchmarks.

Domain (**Variant-1**) or Implementation Optimization in the Code Domain (**Variant-2**) leads to a clear drop in efficiency metrics across all three benchmarks. Furthermore, removing Code Correctness Refinement (**Variant-3**) results in a notable decline in Pass@1. These outcomes are consistent with the design intent: algorithmic exploration and implementation optimization primarily contribute to efficiency, while correctness refinement ensures that the final output maintains functional correctness after efficiency-driven transformations.

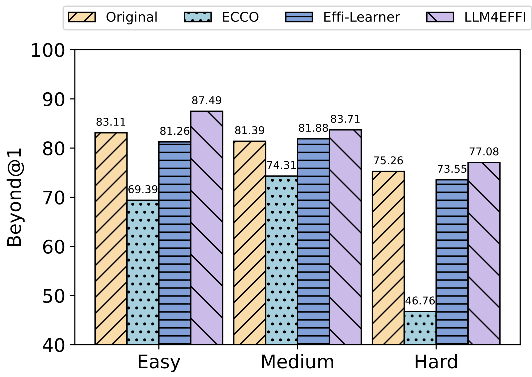

Figure 3: Beyond@1 of LLM4Effi across difficulty levels in Mercury, with DeepSeek-V3 as the backbone.

5 Deeper Analysis

5.1 LLM4Effi Uniqueness Analysis

As discussed in the Introduction, LLM4Effi has two key innovations: **Uniqueness 1:** Separation of efficiency optimization into logic and code domains, and **Uniqueness 2:** Reordering the optimization of correctness and efficiency. To gain a deeper understanding of these unique design choices, we conducted the following experiments:

- **w/o Uniqueness-1:** Instead of separating efficiency optimization into logic and code domains, we prompt the LLM to generate code that is both efficient and correct. Based on the formalized task and generated code, the LLM is then asked to propose possible efficiency optimization strategies. The code is refined accordingly, with subsequent steps identical to those in LLM4Effi. The key difference between **w/o Uniqueness-1** and ECCO is that ECCO generates optimization suggestions solely from the code, whereas **w/o Uniqueness-1** leverages both the formalized task and generated code to produce efficiency-focused strategies, followed by correctness refinement.
- **w/o Uniqueness-2:** This variant reverses the prioritization of correctness and efficiency. In this experiment, it first generates code based on the formalized task and refines it for correctness. Then, algorithm exploration is performed based on the formalized task and refined code, followed by optimization using implementation strategies.

The results of the uniqueness analysis are shown in Tab. 3. Without **Uniqueness-1**, performance of Qwen2.5-Coder-32B-Instruct on Mercury showed a slight increase in Pass@1, but in other cases, the performance declined, particularly in terms of DPS_norm on EvalPerf and eff@1 on ENAMEL. This highlights the value of separating efficiency optimization into logic and code domains, as it decomposes the complex challenge of code efficiency into manageable and focused stages. On the other hand, removing **Uniqueness-2** resulted in notable drops in both efficiency and correctness across all benchmarks and LLM backbones. This is primarily because prioritizing correctness before efficiency limits the LLM's ability to explore effective algorithmic and implementation strategies. In practice, this reversal often backfires: overemphasizing efficiency after correctness can inadvertently compromise correctness itself. These findings underscore the effectiveness of the "efficiency-first, correctness-later" strategy as a critical paradigm for generating code that is both correct and efficient.

Methods	BigCodeBench-Hard	
	Pass@1	Relative Time Cost
Instruct	32.43	1
ECCO	15.54	-11.58% (Faster)
Effi-Learner	25.68	+11.98% (Slower)
LLM4EFFI	**36.50**	**-33.37%** (Faster)

Table 4: Performance of LLM4EFFI vs. Baselines on BigCodeBench-Hard, using DeepSeek-V3 as backbone.

Methods	EvalPerf		ENAMEL	
	DPS_norm	Pass@1	eff@1	Pass@1
Effi-Learner	79.00	88.14	52.22	83.80
Effi-Learner with synthetic test cases	80.16	87.29	52.98	87.32
LLM4EFFI	87.08	90.67	60.41	89.44

Table 5: Performance comparison of LLM4EFFI and Effi-Learner with synthetic test cases, using DeepSeek-V3 as the LLM backbone.

5.2 Performance on Varying Difficulty Levels

To assess LLM4EFFI's performance across different difficulty levels, we evaluate it in three categories: Easy, Medium, and Hard, using Mercury benchmark with DeepSeek-V3 as the backbone. As shown in Fig. 3, LLM4EFFI consistently achieves the highest Beyond@1 scores, outperforming all baselines across difficulty levels. This consistent performance highlights LLM4EFFI 's effectiveness in handling a broad range of coding challenges. In contrast, ECCO exhibits a significant drop on hard-level tasks, primarily due to the difficulty of generating valid optimization suggestions for complex code, a persistent challenge for current LLMs.

5.3 Generalization to More Complex Tasks

To further assess the effectiveness and scalability of LLM4EFFI in more complex software engineering scenarios, we conducted additional experiments using the BigCodeBench (Zhuo et al., 2025) benchmark, a comprehensive suite designed to evaluate the code generation capabilities of LLMs in real-world, large-scale programming tasks. Specifically, we selected the BigCodeBench-Hard subset, which consists of 148 high-difficulty tasks. All other experimental settings remained consistent with the main experiments. In the evaluation, we normalized the execution time of code generated via direct instruction prompting as the baseline (set to 1) and computed the relative time costs of different methods. Results are shown in Table 4. We observe that methods such as ECCO and Effi-Learner, which adopt a "generate-then-optimize" paradigm, suffer significant drops in accuracy. Notably, Effi-Learner often produces less efficient code under complex tasks, leading to longer execution times. In contrast, LLM4EFFI maintains robust and consistent performance, improving code efficiency by 33.37% while also enhancing correctness. These results demonstrate LLM4EFFI's effectiveness in handling challenging, real-world programming tasks.

5.4 Effectiveness of Synthetic Test Cases

In its original design, Effi-Learner relies on oracle test cases. To further assess the effectiveness of LLM4EFFI's synthetic test cases, we conducted an additional comparison with Effi-Learner using synthetic test cases. The results are shown in Tab. 5. When using synthetic test cases for execution refinement, Effi-Learner achieves efficiency metrics (DPS_norm and eff@1) comparable to, and in some cases slightly better than, its original version using oracle test cases. This is largely because the synthetic test cases generated by LLM4EFFI are designed to provide more comprehensive code coverage and finer-grained difficulty differentiation. The Pass@1 results vary across benchmarks, primarily due to the increased difficulty of the synthetic test cases. In more challenging benchmarks such as ENAMEL, higher-difficulty test cases lead to more significant improvements in correctness.

5.5 Case Study and Computational Cost

To provide a clearer understanding of LLM4EFFI, we present a case study, detailed in Appendix B. Methods like ECCO and Effi-Learner are constrained by the algorithmic design and structure of the initial code. In contrast, LLM4EFFI overcomes these limitations by enabling high-level algorithmic exploration from the outset, allowing it to achieve more substantial improvements in code efficiency. Additionally, we provide the computational costs of LLM4EFFI and the baselines in Appendix C.

6 Conclusion

In this paper, we introduce LLM4EFFI, a novel framework for efficient code generation. By decoupling efficiency optimization into logic and code domains and adopting an "efficiency-first, correctness-later" paradigm, LLM4EFFI enables broader algorithmic exploration while ensuring correctness. Extensive experimental results demonstrate LLM4EFFI 's effectiveness and robustness.

Limitation

Although LLM4Effi demonstrates strong performance in generating both efficient and correct code, it still faces certain limitations. One key challenge is balancing code efficiency with readability. To optimize runtime performance, the system often employs complex algorithms or heavily optimized built-in functions. While this improves efficiency, it can also increase cognitive load, particularly for users with limited programming experience, thereby raising the barrier to understanding and applying the generated code in practical scenarios. Improving the readability of LLM-generated code is therefore an important direction, as it can help lower the entry threshold for users. However, this remains a relatively independent research problem, which we plan to investigate further in future work.

References

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

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2024. Improving code generation by training with natural language feedback. Preprint, arXiv:2303.16749.

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

Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, page 79–89. ACM.

DeepSeek-AI. 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. Preprint, arXiv:2405.04434.

DeepSeek-AI. 2024b. Deepseek-v3 technical report. Preprint, arXiv:2412.19437.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. In The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

Hakan Erdogmus, Grigori Melnik, and Ron Jeffries. 2010. Test-driven development. In Encyclopedia of Software Engineering.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. ArchCode: Incorporating software requirements in code generation with large language models. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 13520–13552, Bangkok, Thailand. Association for Computational Linguistics.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. 2024a. Enhancing large language models in coding through multi-perspective self-consistency. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1429–1450, Bangkok, Thailand. Association for Computational Linguistics.

Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. 2024b. Effilearner: Enhancing efficiency of generated code via self-optimization. Preprint, arXiv:2405.15189.

Dong HUANG, Yuhao QING, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. In The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

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

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

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

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In The Twelfth International Conference on Learning Representations.

Bin Lei, Yuchen Li, and Qiuwu Chen. 2024. Autocoder: Enhancing code large language model with AIEV-Instruct. Preprint, arXiv:2405.14906.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024a. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. Preprint, arXiv:2404.00599.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024b. Codetree: Agent-guided tree search for code generation with large language models. Preprint, arXiv:2411.04329.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In Thirty-seventh Conference on Neural Information Processing Systems.

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

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In Thirty-seventh Conference on Neural Information Processing Systems.

Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On evaluating the efficiency of source code generated by llms. Preprint, arXiv:2404.06041.

OpenAI. 2024. Gpt-4o system card. Preprint, arXiv:2410.21276.

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

Jieke Shi, Zhou Yang, and David Lo. 2024. Efficient and green large language models for software engineering: Literature review, vision, and the road ahead. Preprint, arXiv:2404.04566.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. Preprint, arXiv:2303.11366.

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

Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In Findings of the Association for Computational Linguistics: EMNLP 2023, pages 1741–1788, Singapore. Association for Computational Linguistics.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. arXiv preprint arXiv:2312.11805.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. 2024. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 15362–15376, Miami, Florida, USA. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with OSS-instruct. In Proceedings of the 41st International Conference on Machine Learning, volume 235 of Proceedings of Machine Learning Research, pages 52632–52657. PMLR.

Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. 2023. Fine-grained human feedback gives better rewards for language model training. In Thirty-seventh Conference on Neural Information Processing Systems.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024a. Qwen2 technical report. arXiv preprint arXiv:2407.10671.

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

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Findings of the Association for Computational Linguistics: ACL 2024, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In The Thirteenth International Conference on Learning Representations.

A Detailed Prompts

A.1 Prompts of LLM4EFFI.

Task Formalization

System:
As a professional algorithm engineer, please analyze the given algorithm problem according to the following categories. Do not provide any example implementation:

- Entry Point Function Name
- Input/Output Conditions
- Edge Cases and Parameter Types (Int, String, etc.)
- Expected Behavior

User:
The algorithm problem description is as follows:
<natural language description>

Figure 4: Task Formalization.

Task Formalization Check

System:
As an excellent algorithm engineer, please analyze whether the explanation of the problem matches the original requirements. If they are consistent, output "Yes". If they are not consistent, output "No" and provide the reason, as shown below: {"Yes":"NULL"} {"No":"The reason is"}
User:
<natural language description>
<task description>

Figure 5: Checking the Task Formalization Result.

Synthesize Test Case Inputs

System:
As a tester, your task is to create comprehensive test inputs for the function based on its definition and docstring. These inputs should focus on edge scenarios to ensure the code's robustness and reliability. Please output all test cases in a single line, starting with input.
User:
EXAMPLES:
Function:

```
from typing import *
def find_the_median(arr: List[int]) ->
    float:
    Given an unsorted array of
        integers `arr`, find the
        median of the array.
    The median is the middle value in
        an ordered list of numbers.
    If the length of the array is
        even, then the median is the
        average of the two middle
        numbers.
```

Test Inputs (OUTPUT format):
input: [1]
input: [-1, -2, -3, 4, 5]
input: [4, 4, 4]
input: [....]
input: [....]
END OF EXAMPLES.
Function:
<task description>

Figure 6: Synthesize Test Case Inputs.

Implementation Optimization in Code Domain

System:
As a professional Python algorithm programming expert, please provide suggestions for improving code efficiency based on the potential inefficiencies mentioned above. For example:
1. Using xxx instead of xxx can significantly improve code efficiency.
Please provide at least 20 suggestions.
User:
<algorithm description>

Figure 7: Implementation Optimization in Code Domain.

Complete Test Case Generation

System:
As a programmer, your task is to calculate all test outputs and write the test case statement corresponding to the test input for the function, given its definition and docstring. Write one test case as a single-line assert statement.
User:
EXAMPLES:
Function:

```
from typing import List
def find_the_median(arr: List[int]) ->
     float:
    Given an unsorted array of
         integers `arr`, find the
         median of the array. The
         median is the middle value in
         an ordered list of numbers.
    If the length of the array is
         even, then the median is the
         average of the two middle
         numbers.
```

Test Input:
input: [1, 3, 2, 5]
Test Case:

```
assert find_the_median([1, 3, 2, 5]) == 2.5
```

END OF EXAMPLES.
FUNCTION:
<task description> <input case>

Figure 8: Complete Test Case Generation.

Algorithmic Exploration in Logic Domain

System:
As a professional algorithm engineer, you can effectively design multiple algorithms to solve the problem with low time complexity and output them in pseudo algorithm format. A pseudo algorithm is a nonlinear, high-level programming language for algorithmic logic. It combines natural language and programming structures to express the steps and sums of algorithms. The main purpose of process algorithms is to clearly display the core ideas and logic of the algorithm without relying on specific programming language syntax. Please design 5 excellent algorithm solutions based on the problem description provided. The time complexity of the algorithm needs to be as small as possible, and try to output 5 algorithms in the form of a pseudo-algorithm in the following format: PS: DO NOT provide implementation examples!

````
```algorithm1
{algorithm key description: this
     algorithm using xxx, the key is to
     make sure xxx}
{pseudo algorithm: ..}

{algorithm key description: this
     algorithm using xxx, the key is to
     make sure xxx}
{pseudo algorithm: ..}

{algorithm key description: this
     algorithm using xxx, the key is to
     make sure xxx}
{pseudo algorithm: ..}

{algorithm key description: this
     algorithm using xxx, the key is to
     make sure xxx}
{pseudo algorithm: ..}

{algorithm key description: this
     algorithm using xxx, the key is to
     make sure xxx}
{pseudo algorithm: ..}
````

User:
<task description>

Figure 9: Algorithmic Exploration in Logic Domain.
````

Code Candidates Generation

System:
As a professional algorithm engineer, please convert the selected algorithm into corresponding code. Ensure the code is complete and well-formatted. When converting to a standardized format, be sure to follow the guidelines specified in the "original question format":
1. Use the same function name as given in the original question format; do not rename it.
2. You may incorporate practical optimization details drawn from the knowledge base.
The final output format should be as follows:

```python
{<code>
```

User:
<task description>
<algorithm description>
<efficiency optimization suggestions>

Figure 10: Code Candidates Generation.

Code Refinement for Correctness

System:
As a professional code programming algorithm expert, your task is to correct the code and ensure that the code is fixed without impacting its time complexity or practical efficiency. Then I will provide you with specific code and test cases.
Important Notes:
1. Do not alter the algorithm itself
2. Do not change the format, such as the function name.
3. Please output in the specified format.
4. Ensure there are no syntax errors.
Please output in this format:

```python
{code}
```

User:
<task description>
<algorithm description>
<efficiency optimization suggestions>

Figure 11: Code Refinement for Correctness.

Final Results Selection on Code Candidates

System:
As a professional algorithm engineer, please help me choose the most efficient code from the following codes. It is worth mentioning that it is necessary to consider the time complexity and practical level comprehensively:
INPUT:
{ "1":"def ...()....",
"2": "def ...()..."
}
OUTPUT:

```text
{key}
```

EXAMPLE:
INPUT:
{ "1":"def ...()....",
"2": "def ...()..."
}
OUTPUT:

```text
1
```

User:
<corrected code candidate>

Figure 12: Final Results Selection on Code Candidates (Optional).

Direct Code Generation Prompt for Variant-1

System:
As a professional Python algorithm engineer, please solve the algorithms problem and generate a solution code. The final output format should be as follows:

```python
{code}
```

User:
<task description>

Figure 13: Direct Code Generation Prompt for Variant-1.

Direct Code Generation Prompt for w/o Uniqueness-1&w/o Uniqueness-2

System:
As a professional Python algorithm engineer, please solve the algorithm problem and generate 5 solution codes. Please improve the efficiency of the code as much as possible while ensuring the correctness of the code. The final output format should be as follows:

````
```python1
{code}
```
```python2
{code}
```
```python3
{code}
```
```python4
{code}
```
```python5
{code}
```
````

User:
<task description>

Figure 14: Direct Code Generation Prompt for w/o Uniqueness-1&w/o Uniqueness-2.

A.2 Prompts of Effi-Learner.

Original Code Generation Prompt in Effi-Learner

Please complete Python code based on the task description.
Task description:<Task description>
#Solution:

Figure 15: Original Code Generation Prompt in Effi-Learner.

Efficiency Optimization Prompt in Effi-Learner.

Optimize the efficiency of the following Python code based on the task, test case, and overhead analysis provided. Ensure the optimized code can pass the given test case.
Task Description:
<task description>
Test Case:
<test case>
Original Code:

````
```python
<original code>
```
````

Overhead Analysis:
<profile of original code>
Optimization Rules:
- Encapsulate the optimized code within a Python code block (i.e., python[Your Code Here]).
- Do not include the test case within the code block.
- Focus solely on code optimization; test cases are already provided.
- Ensure the provided test case passes with your optimized solution.

Figure 16: Efficiency Optimization Prompt in Effi-Learner.

A.3 Prompts of ECCO.

Original Code Generation Prompt in ECCO

Write a python code which is efficient in terms of runtime and memory usage for the following problem description. Wrap the optimized code in a block of 3 backticks

Figure 17: Original Code Generation Prompt in ECCO.

Feedback Generation Prompt in ECCO

Give feedback in english for why the code solution below is incorrect or inefficient and how the program can be fixed based on the problem description.
<original code>

Figure 18: Feedback Generation Prompt in ECCO.

DeepSeek-V3	Tokens on Synthetic Test Case		Tokens on Framework		Cost
	Test Case Input	Test Case Output	Input tokens to LLM	LLM output tokens	
ECCO	-	-	$\sim$3730 tokens	$\sim$1027 tokens	$\sim$0.0014$
Effi-Learner	-	-	$\sim$1241 tokens	$\sim$364 tokens	$\sim$0.0006$
LLM4EFFI	$\sim$2612 tokens	$\sim$818 tokens	$\sim$8170 tokens	$\sim$4351 tokens	$\sim$0.0096$

Table 6: Computational cost of LLM4EFFI and baseline methods using DeepSeek-V3 as the LLM backbone.

Refine Prompt in ECCO

Refine the given incorrect or sub-optimal code solution based on the feedback specified below. Wrap the refined code in a block of 3 backticks
<optimization suggestion>
<original code>

Figure 19: Refine Prompt in ECCO.

A.4 Prompts of Instruct.

Prompt for Instruction Baseline

Please generate an efficient and correct code directly

Figure 20: Prompt for **Instruction** Baseline.

B Case Study

B.1 The Execution Details of Each Stage of LLM4EFFI.

As shown in Fig. 21, LLM4EFFI firstly analyzes the algorithm problem, *"returns the n-th number that is both a Fibonacci number and a prime number"*, providing a detailed explanation of key aspects, including the entry point, input/output conditions, expected behavior, and edge cases. Based on this analysis and the problem description, LLM4EFFI explores potential algorithms and generates five efficient solutions, such as using the Fibonacci sequence generation method and Binet's formula. Next, LLM4EFFI examines the implementation details of these algorithms and identifies the optimal practical approaches. For example, it uses Python's built-in `pow()` function for efficient exponentiation and applies the Miller-Rabin primality test (based on Monte Carlo method) to enhance the efficiency of prime number detection for large numbers.

Then, LLM4EFFI combines the explored algorithms and practical operations to generate five distinct code implementations. To validate the correctness of these codes, LLM4EFFI generates 20 test cases based on the algorithm description and outputs them in the format *"assert prime_fib(3) == 5"*. Each code candidate is then executed with these 20 test cases, recording the number of passed test cases ($Pass_t \leq 20$) and the number of successful executions for each test case ($Pass_c \leq 5$). Subsequently, LLM4EFFI checks the test cases that are not passed by the code implementations, ensuring that correct test cases are not excluded due to code errors and preventing incorrect test cases from being misused in subsequent iterations.

After filtering, LLM4EFFI generates a new batch of test cases and executes them again to obtain updated results. For the test cases that fail, an iterative feedback mechanism is applied to optimize the code. The code, now enhanced with this iterative feedback, is executed once more, and the final passing results are recorded. All codes are then ranked in descending order of correctness, with the most accurate code being selected. This process ensures the identification of the optimal solution, while maintaining both high efficiency and accuracy in the code implementation.

B.2 Comparison of Methods.

In Fig. 22 and Fig. 23, we compare the code efficiency optimization processes of LLM4EFFI and baselines.

C Computational Cost

To evaluate the computational cost, we conducted additional experiments to measure the total number of generated tokens for LLM4EFFI. Specifically, we calculated the average total number of tokens for each problem task using various approaches under DeepSeek-V3 backbone. The results are shown in Tab.6, with the "Cost" column calculated based on DeepSeek's official pricing documentation.

- LLM4EFFI is the only method that requires the generation of synthetic test cases. This is because LLM4EFFI is designed to simulate more realistic real-world scenarios, where public test cases are often unavailable. In contrast,

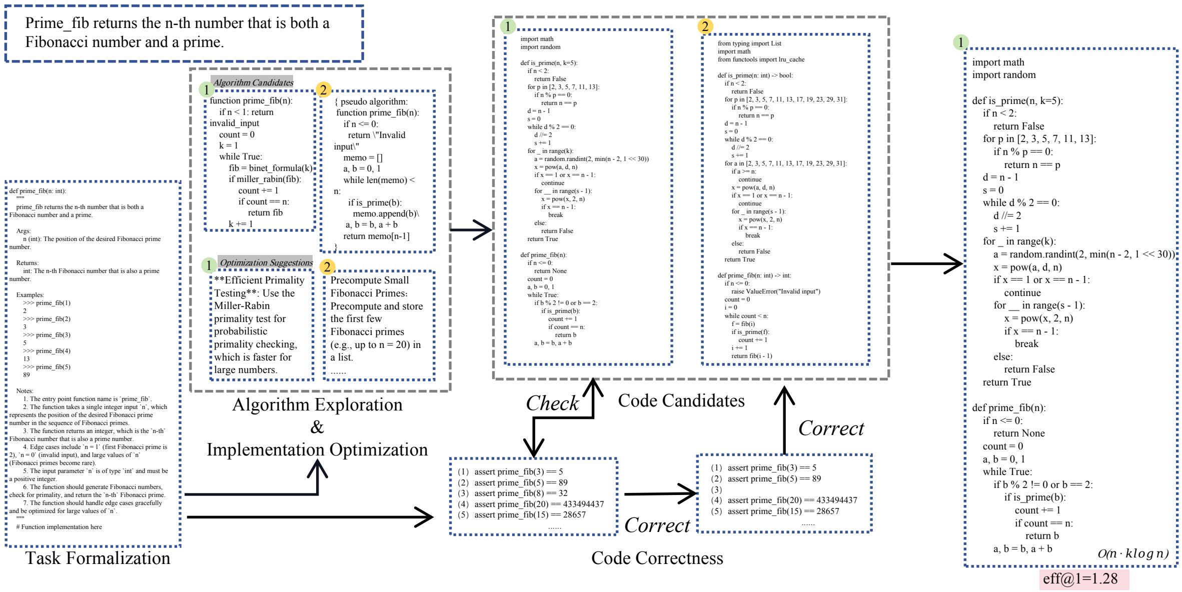

Figure 21: The figure illustrates the specific output of each subtask process of LLM4EFFI in solving algorithm problems.

the baseline method, Effi-Learner, relies on test case oracles provided with the problems.

- LLM4EFFI indeed requires more generated tokens from LLMs compared to baseline methods. However, considering the significant performance gains achieved, we believe the cost remains within an acceptable range.

- LLM4EFFI offers significant flexibility. Users can customize and adjust various parameters, such as the number of synthetic test cases and algorithm explorations, to optimize token usage and reduce associated costs.

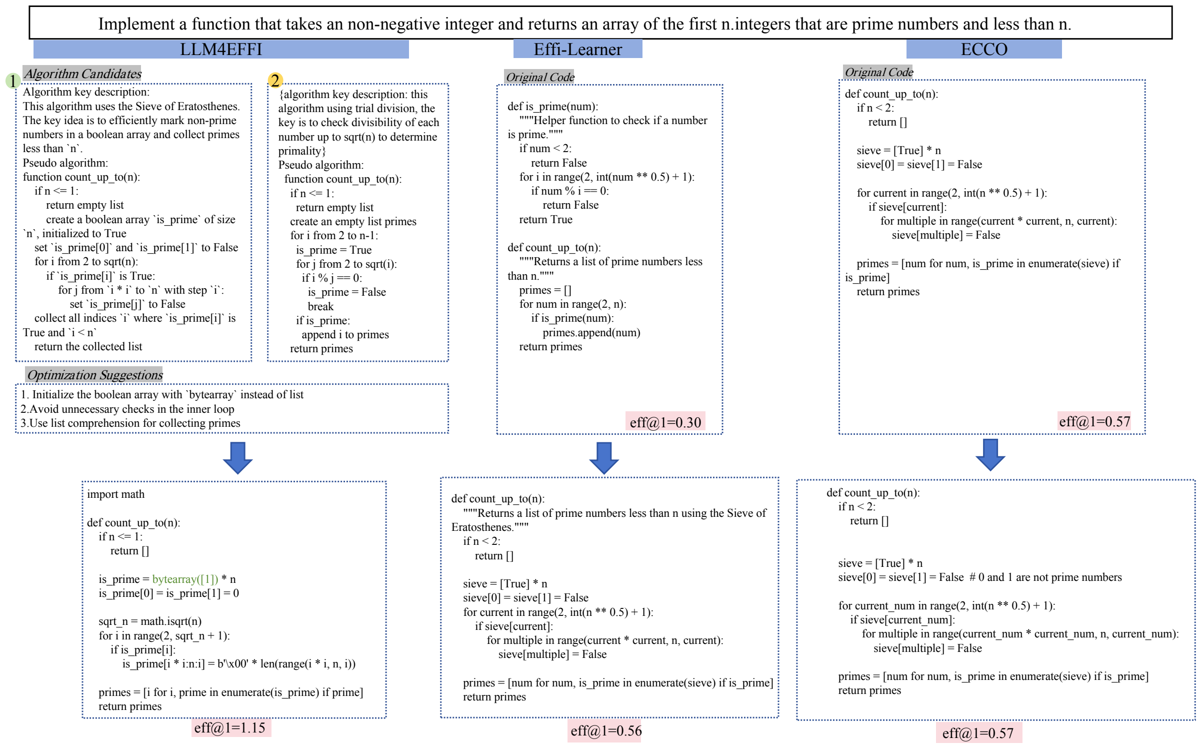

Figure 22: The diagram demonstrates how LLM4EFFI, Effi-Learner, and ECCO generate code. LLM4EFFI, through deep exploration of the algorithm domain, generates a set of efficient and high-quality algorithm candidates. However, the time complexity of these algorithms is similar, and there is no significant difference from the original code generated by Effi-Learner and ECCO. Subsequently, LLM4EFFI identifies key optimization suggestions in its practical recommendations, such as replacing list with bytearray, among others. As a result, although the final code has a similar time complexity to the other two tools, it significantly outperforms them in the final ENAMEL efficiency evaluation metrics.

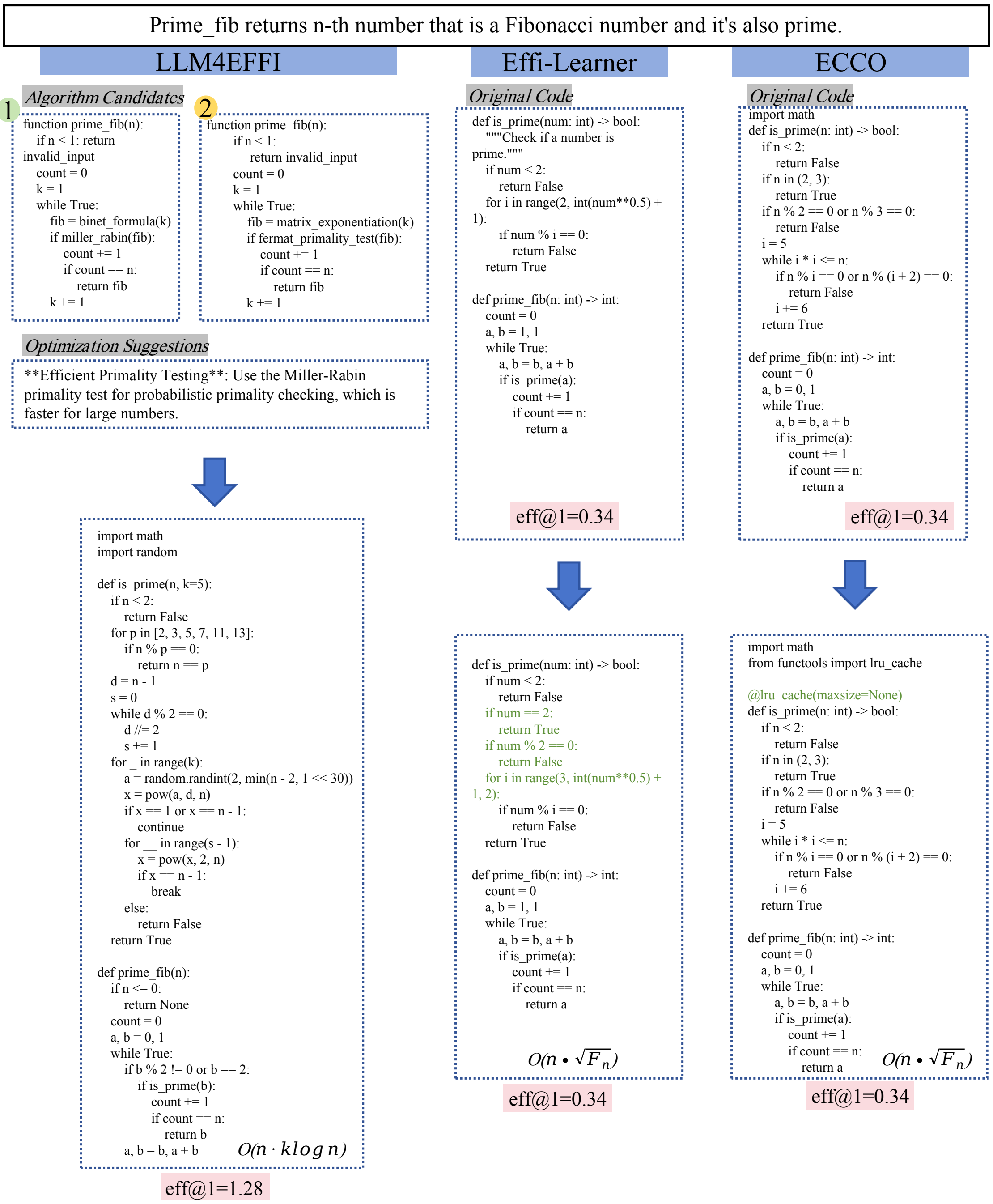

Figure 23: The figure illustrates the code generation process of LLM4EFFI, Effi-Learner, and ECCO. LLM4EFFI, through deep exploration of the algorithm domain, generates a set of efficient and high-quality algorithm candidates. By incorporating practical optimization suggestions, it ultimately produces an algorithm with a time complexity of only $\mathcal{O}(n \cdot k \log n)$, achieving a high score of 1.28 on the ENAMEL test set. In contrast, Effi-Learner and ECCO, constrained by the $\mathcal{O}(n \cdot \sqrt{F_n})$ time complexity of their code algorithms, can only perform local optimizations on certain implementations, resulting in minimal improvements, with the final efficiency index reaching only 0.34.