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

Anonymous ACL submission

Abstract

Large Language Models (LLMs), particularly Code LLMs, have demonstrated impressive performance in code generation. Current research primarily focuses on the correctness of generated code, while efficiency remains less explored. Recent works have focused on modifying the initial version of the code to improve its efficiency. However, such refinements are limited by the algorithmic design and overall logic of the initial code, resulting in only incremental improvements. In contrast, when human developers write high-quality code, they typically begin by designing several potential solutions at the logical level, evaluating various algorithms and their complexities, and then proceeding to implement and optimize the solution. In this study, we introduce LLM4EFFI: Large Language Model for Code Efficiency, a novel framework that enables LLMs to generate code that balances both efficiency and correctness. Specifically, LLM4EFFI divides the efficiency optimization process into two domains: algorithmic exploration in the logic domain and implementation optimization in the code domain. The correctness of the code is then guaranteed through a synthetic test case refinement process. This approach, which prioritizes efficiency before ensuring correctness, offers a new paradigm for efficient code generation. Experiments demonstrate that LLM4EFFI consistently improves both efficiency and correctness, achieving new state-of-the-art performance in code efficiency benchmarks across various LLM backbones.

1 Introduction.

Large Language Models (LLMs), particularly those specialized in code, are revolutionizing the field of software engineering at an unprecedented pace. A significant area of advancement lies in automated code generation (Liu et al., 2023), 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) demonstrating remarkable capabilities. These models have attracted considerable attention from both academia and industry, consistently breaking new ground on code completion and generation benchmarks, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCodeBench (Jain et al., 2024).

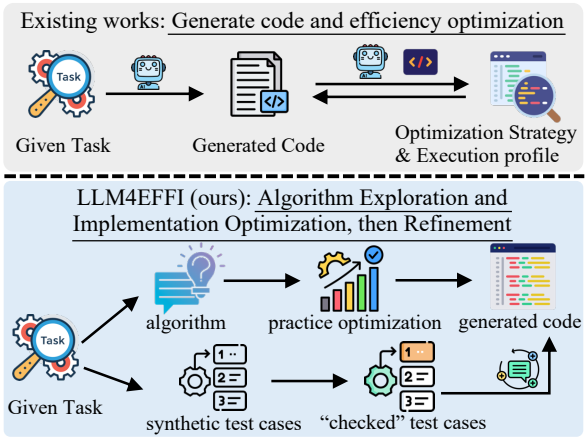

Figure 1: Comparison of LLM4EFFI with existing methods. Existing methods generate code first, then optimize it using strategy and execution profiles. In contrast, LLM4EFFI starts with the task, focusing on efficiency through algorithm exploration and implementation, followed by correctness refinement.

While these LLMs achieve impressive accuracy in automatic code generation, practical software engineering applications require more than just correct code—they also demand efficiency (Shi et al., 2024; Niu et al., 2024). In real-world scenarios, even correct code often requires manual optimization by engineers before it can be used, which undermines the goal of "out-of-the-box" automated code generation. Therefore, generating code that is both correct and efficient is essential, yet automating this process has not been widely explored.

Recent preliminary works (Huang et al., 2024b; Waghjale et al., 2024) have explored feedback-

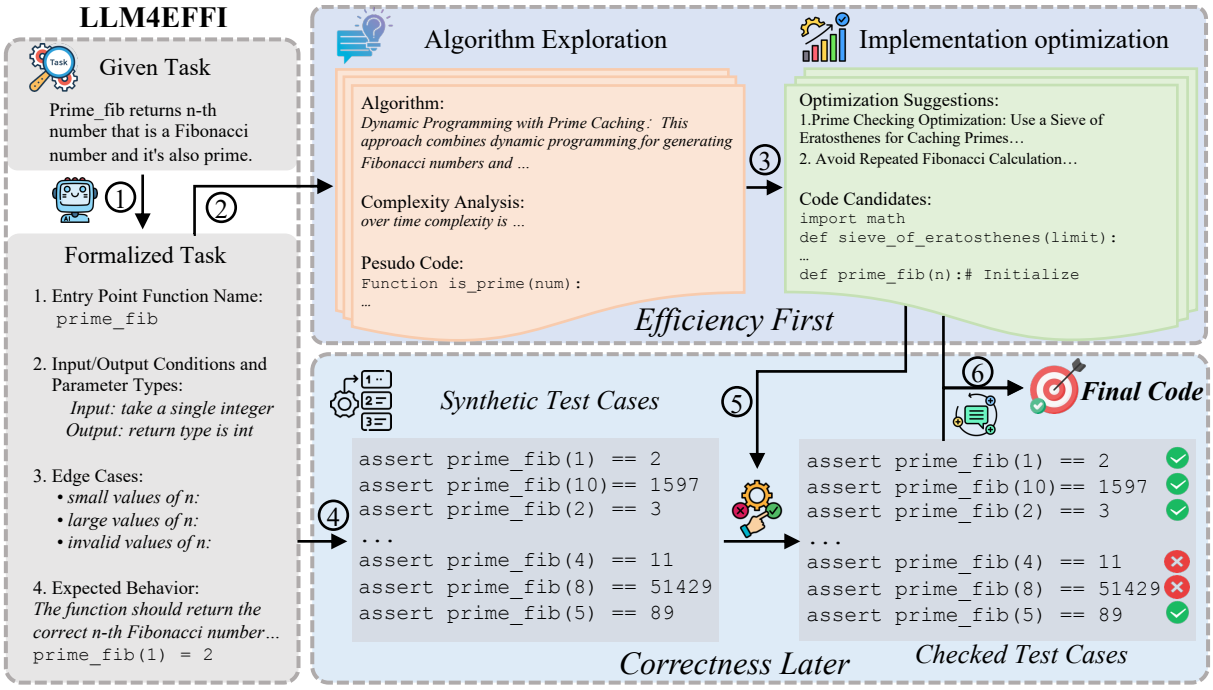

Figure 2: The workflow of LLM4EFFI. Given a programming task, LLM4EFFI formalizes it into a code-oriented description, generates optimal algorithms and pseudocode in logic domain, and then produces implementation suggestions in code domain. LLM4EFFI synthesizes test cases and uses a verification-based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

based approaches to optimize generated code and enhance its efficiency. As illustrated in Figure 1, these methods typically involve profiling code execution time and incorporating reflective feedback into the optimization process. However, the "generate-then-optimize" paradigm is constrained by the algorithmic design and overall structure of the initial code, leading to only incremental improvements. We provide detailed examples in Figure 23 and 22 in Appendix B. In contrast, when human developers write high-quality code, whether in practical software development or algorithmic teaching scenarios, they typically start by designing multiple potential solutions at a logical level. For example, when tackling a sorting problem, a developer might consider Quicksort for its average-case efficiency of $\mathcal{O}(N * \log N)$, while also factoring in its worst-case time complexity of $\mathcal{O}(N^2)$. By carefully analyzing the problem's constraints and evaluating various algorithms along with their complexities, they then proceed to implement the solution, applying various coding techniques to optimize it. Finally, they debug and refine the code to achieve a high-quality implementation.

Inspired by this thought, we propose LLM4EFFI, as shown in Figure 2, a novel paradigm that enables LLMs to generate both efficient and correct code. Specifically, for a given programming task described in natural language, LLM4EFFI first "formalizes" it into a code-oriented problem description. In other words, it converts the broad natural language statement into a clear, concrete, and well-defined coding problem, ensuring that the LLM can accurately interpret it. Next, LLM4EFFI prompts the LLM for logic-level reasoning and exploration, considering various algorithmic approaches, providing corresponding complexity analyses, and generating relevant pseudocode. Based on these different algorithm designs and their associated pseudocode, LLM4EFFI suggests code implementation strategies, followed by code generation and optimization at the implementation level, as high-quality code also requires careful consideration during the practical implementation stage. To ensure the functional correctness of the generated code while targeting efficiency, LLM4EFFI introduces a bidirectional verification-based adaptive testing framework to check synthetic test cases. Finally, the code solutions are executed on the "checked" test cases and iterated upon for correctness. The solution with highest pass rate across the "checked" test cases is selected as the final generated code.

The LLM4EFFI has two distinctive uniqueness:
Uniqueness 1: Separation of Efficiency Optimiza-

tion into Logic and Code Domains. LLM4EFFI divides efficiency optimization into two distinct domains: the "logic domain" and the "code domain". In the logic domain, efficiency optimization focuses on exploring the optimal algorithmic approaches, while in the code domain, optimization deals with the practical implementation details. This separation effectively breaks down the challenge of optimizing code efficiency into manageable steps, making the overall efficiency optimization process more systematic and targeted.

Uniqueness 2: The Order of Correctness and Efficiency. The order in which correctness and efficiency are optimized plays a critical role. By prioritizing efficiency first, a wider range of algorithmic solutions can be explored, leading to the discovery of multiple efficient approaches. Correctness is then incrementally ensured across these solutions. This approach avoids prematurely constraining efficiency optimization by focusing too early on correctness. Prioritizing efficiency first allows for greater room for improvement and significantly enhances the potential for efficiency gains.

We validate 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 show that LLM4EFFI consistently enhances both code correctness and efficiency across various LLM backbones, achieving state-of-the-art performance in efficiency metrics. Specifically, using the DeepSeek-V3 backbone, LLM4EFFI improves eff@1 by 9.27% on ENAMEL and boosts DPS_norm by 6.63% on Mercury.

Overall, we summarize our contributions as follows, with corresponding code available at link[1]:

- We propose LLM4EFFI, the first framework that simultaneously optimizes both code efficiency and correctness.
- We introduce two key features: *Separation of Efficiency Optimization into Logic and Code Domains* and *Order of Correctness and Efficiency*. We hope these unique features will contribute to the advancement of the code efficiency community.
- Extensive experiments and analysis on three benchmarks across different LLM backbones demonstrate the effectiveness and robustness of LLM4EFFI in efficient code generation.

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

2 Related Works

2.1 LLMs for Code Domain.

Large language models have been widely applied to coding tasks and have shown strong performance across various coding scenarios and evaluations. Most existing research focuses on code generation, with numerous techniques developed to enhance its quality. Some methods aim to improve the quality of synthetic code data (Wei et al., 2024; Luo et al., 2024; Lei et al., 2024), enhance self-consistency (Le et al., 2024; Huang et al., 2024a), or leverage 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., 2024). However, these methods primarily focus on the correctness of the generated code, with relatively little emphasis on the efficiency of the generated code.

2.2 Code Efficiency.

Until recently, the academic community has only begun to pay significant attention to the efficiency of generated code. Recently, several efficiency-focused benchmarks (HUANG et al., 2024; Du et al., 2024; Liu et al., 2024; Qiu et al., 2024) have emerged, aiming to provide a more comprehensive evaluation of LLMs' ability to generate efficient code. However, empirical studies and evaluations of these benchmarks show that current LLMs still face significant challenges in generating efficient code. To improve code efficiency, recent research such as ECCO (Waghjale et al., 2024) adopts self-refinement, prompting LLMs to consider possible optimization strategies and refine their outputs. Effi-Learner (Huang et al., 2024b) proposes a self-optimization framework that uses execution overhead profiles, feeding them back into the LLM to revise the code and reduce time overhead. However, these methods focus on enhancing the efficiency of code after it has been generated, 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 sample is represented as a pair (Q, T_h), where Q denotes the task description, and T_h corresponds to the hidden test cases. Our goal is to generate the corresponding code solution S that passes

the hidden test cases and achieves the highest efficiency (i.e., the shortest execution time). Notably, to better simulate real-world scenarios, we assume there are no public test cases. T_h is only used during the evaluation stage and is not visible during efficiency and correctness optimization stages.

3.1 Overview.

We present the framework of LLM4EFFI in Figure 2. For a given programming task described in natural language, LLM4EFFI first "formalizes" it into a code-oriented problem description (**Section 3.2**). Next, LLM4EFFI queries the LLM for logic-domain reasoning and exploration, generating multiple optimal algorithmic solutions along with their corresponding pseudocode (**Section 3.3**). Based on these algorithm designs and their associated pseudocode, LLM4EFFI analyzes and generates code implementation suggestions, followed by the generation and optimization of the corresponding code at the implementation level (**Section 3.4**). To further refine the solutions for correctness, LLM4EFFI synthesizes a large number of test cases and utilizes a bidirectional verification-based adaptive testing framework to "check" these synthetic test cases. The "checked" test cases are then used to evaluate the candidate code solutions (**Section 3.5**). The solution with highest pass rate across the "checked" test cases is selected as the final generated code.

3.2 Task Formalization.

In the initial task formalization stage, LLM4EFFI ensures that the task description is clear and unambiguous, which is crucial for the success of 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 comprehend the task from four key dimensions: entry point function name, input/output conditions and parameter types, edge cases, and expected behavior. Based on these dimensions, the LLM is further encouraged to engage in self-reflection to confirm whether it has fully grasped all aspects of the task, thus laying a solid foundation for the subsequent stages. Formally, $Q \rightarrow Q_{formal} \overset{\text{check}}{\longleftrightarrow} Q$.

3.3 Algorithmic Exploration in Logic Domain.

For the formalized task defined in the first stage, LLM4EFFI prompts the LLM to engage in algorithmic reasoning at the logical level, rather than immediately generating code. This approach mirrors that of human programmers, who first perform abstract and high-level reasoning before implementation. The LLM is prompted to explore multiple potential optimal algorithms, analyze their corresponding complexities, and represent the entire logical process with pseudocode. Formally, $Q_{formal} \rightarrow \{Algo, Cplx, Pseudo\}$, where $Algo$ refers to the algorithm plan, $Cplx$ refers to the complexity analysis, and $Pseudo$ refers to the corresponding pseudocode.

3.4 Implementation Optimization in Code Domain.

Excellent code not only requires careful algorithm design but also necessitates optimization at the implementation level. Even when the same algorithm is used, different implementation approaches can lead to significant variations in code efficiency (Shypula et al., 2024; Coignion et al., 2024). When implementing code based on the algorithm plan and corresponding pseudocode, LLM4EFFI prompts the LLM to provide practical suggestions derived from $Algo$ and $Pseudo$, such as replacing a manual binary exponentiation implementation with Python's built-in pow function, among other optimizations. We provide three detailed examples in the appendix to illustrate this process. Subsequently, LLM4EFFI generates the corresponding code based on the $Algo$, $Pseudo$, and implementation suggestions, while also checking for further optimization opportunities. Formally, $\{Algo, Pseudo\} \rightarrow \{Suggs\}$, and $\{Algo, Pseudo, Suggs\} \rightarrow \{Code\ Candidates\}$.

3.5 Code Correctness.

To ensure the functional correctness of generated code while targeting efficiency, LLM4EFFI introduces a bidirectional verification-based adaptive testing framework. The process works as follows: First, LLM4EFFI 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, thoroughly testing the robustness and reliability of the generated code. However, since the synthesized test cases may not be entirely correct, LLM4EFFI performs bidirectional verification to validate them.

Forward Verification: If all candidate code implementations pass a specific test case, the test case is marked as trusted. Otherwise, **Reverse Review:**

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.** The results of LLM4EFFI, Instruct, ECCO, and Effi-Learner methods on the EvalPerf, Mercury, and ENAMEL benchmarks are presented using the Qwen2.5-Coder-32B-Instruct, Qwen2.5-72B-Instruct, GPT-4o-mini, GPT-4o, and DeepSeek-V3 LLM backbones. Correctness is evaluated using Pass@1, while efficiency is measured using the respective efficiency metrics for each of the three benchmarks.

For test cases that cause failures in any candidate code, LLM4EFFI performs a Q_{formal}-based review. It checks whether the test case aligns with the intent of the formal task description and conducts a semantic consistency check, which is similar to Test-Driven Development (Erdogmus et al., 2010) in software engineering. If the test case passes the reverse review, it is retained; otherwise, it is discarded. Finally, the retained test cases are marked as "checked". These "checked" test cases are then used to evaluate the generated code candidates, with any failures triggering further refinements. The code candidate that passes the most "checked" test cases is ultimately selected as the final solution. Formally, $Q_{formal} \rightarrow \{Synth.\ test\ cases\} \overset{\text{check}}{\rightarrow} \{Checked\ test\ cases\} \overset{\text{select}}{\rightarrow} \{final\ solution\}$.

4 Experiments

We evaluate LLM4EFFI on three code efficiency evaluation benchmarks: EvalPerf (Liu et al., 2024), Mercury (Du et al., 2024) and ENAMEL (Qiu et al., 2024). **EvalPerf** focuses on performance-challenging tasks and 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 functional 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 code efficiency using the eff@1 metric. This efficiency score is determined by measuring the worst execution time of the code sample across test cases of varying difficulty levels. The score is then adjusted using a weighted average across these levels to account for hardware fluctuations. The eff@1 metric ranges from 0 to 1, with higher values indicating greater code efficiency. A value exceeding 1 signifies that the generated code is more efficient than the expert-level solution.

4.1 Compared Methods.

We evaluate the direct instruction of generating correct and efficient code as the **Instruct** baseline. We compare LLM4EFFI with two recent proposed

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.** The results of LLM4EFFI, Variant-1, Variant-2, and Variant-3 are presented using Qwen2.5-Coder-32B-Instruct and DeepSeek-V3 as LLM backbones on the EvalPerf, Mercury, and ENAMEL benchmarks. We have highlighted the performance changes of Variants compared to LLM4EFFI with colors.

methods **ECCO** (Waghjale et al., 2024) and **Effi-Learner** (Huang et al., 2024b) for code efficiency.

- **ECCO:** A self-refine with NL feedback approach that prompts the LLM to generate code, then asks if improvements in correctness or efficiency can be made, and finally refines the solution based on optimization suggestions.
- **Effi-Learner:** First generates code using instruction prompts same as **Instruct** baseline, then executes the code with test cases to collect performance profiles, including runtime and memory usage. These profiles are fed back into the LLM along with the code, prompting the LLM to refine the code for efficiency based on the profile. It is worth noting that Effi-Learner relies on test case oracles, and in this study, we use the visible test cases from the task. In contrast, LLM4EFFI does not rely on any test case oracles; all test cases are synthetically generated by LLM4EFFI itself.

4.2 Experiment Setup.

To comprehensively evaluate LLM4EFFI, we selected five different LLM backbones: two proprietary models, GPT-4o (OpenAI, 2024) and GPT-4o-mini, and three open-source models, 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 in LLM4EFFI are detailed in Appendix A. To ensure consistency and a fair comparison, all experiments were conducted with the temperature set to 0, and each experiment was repeated three times to compute an average, thereby eliminating any potential disruptions.

4.3 Main Results.

We compare LLM4EFFI with the other methods on the EvalPerf, Mercury, and ENAMEL benchmarks, and present the results in Table 1. First, we observe that direct instruction prompts yield good performance, indicating that LLMs have a reasonable understanding of correct and efficient code. Then, through ECCO, we observe a slight improvement in efficiency on EvalPerf and Mercury. However, this improvement often comes at the cost of correctness. Particularly in more complex benchmarks like ENAMEL, the approach results in a decline in both efficiency and correctness. This suggests that relying solely on code understanding to generate optimization suggestions is insufficient. When optimization strategies are based purely on code-level analysis, they often fail to align with the broader logical requirements of the task. The mismatch between the code domain and the logic strategy domain makes such methods less effective.

Moreover, Effi-Learner shows some gains in efficiency and correctness on specific benchmarks, such as when using GPT-4o on the Mercury benchmark. However, its performance varies significantly across different LLM backbones and benchmarks, often falling short of the direct Instruct baseline. More importantly, Effi-Learner faces a recurring issue: both efficiency and correctness suffer simultaneously. This stems from its feedback mechanism, which focuses solely on performance metrics like execution time, neglecting the code's functionality and correctness. Additionally, the lack of a comprehensive algorithmic strategy leads to an over-prioritization of execution time, often sacrificing code accuracy and resulting in a decline in both efficiency and correctness.

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: **LLM4EFFI Uniqueness Study Results.** The results of LLM4EFFI, **w/o Uniqueness-1**, and **w/o Uniqueness-2** are presented using Qwen2.5-Coder-32B-Instruct and DeepSeek-V3 as LLM backbones on the EvalPerf, Mercury, and ENAMEL benchmarks. We have highlighted the performance changes compared to LLM4EFFI with colors.

In comparison, LLM4EFFI achieves a simultaneous improvement in both correctness and efficiency through efficiency optimizations at the logical and code implementation levels, followed by refinement of correctness using "checked" test case feedback. The results demonstrate that LLM4EFFI delivers robust and consistent performance improvements across various benchmarks and different LLM backbones, with the gains highlighted in color. For example, using DeepSeek-V3 as the backbone on EvalPerf, LLM4EFFI improved the efficiency metric DPS_norm by 6.63%, while on ENAMEL, eff@1 increased by 9.27%, and Pass@1 improved by 2.82%.

4.4 Ablation Study.

LLM4EFFI incorporates several unique design choices, such as separating efficiency optimization into the logic domain and code implementation level. To better understand the impact of each component, we conduct the following ablation study:

- **Variant-1:** (Without Algorithmic Exploration in the Logic Domain): In this variant, no algorithmic exploration is performed for efficiency optimization in the logic domain. Instead, the LLM directly generates the same count efficient code solution, followed by implementation-level optimization (based on the formalized task and the generated code solution). All other steps remain the same as in LLM4EFFI.
- **Variant-2:** (Without Implementation Optimization in the Code Domain): This variant omits the implementation optimization step in the code domain, while all other processes are identical to those in LLM4EFFI.
- **Variant-3:** (Without Code Correctness Refinement): In this variant, after generating the efficiency-optimized code solutions, the LLM independently selects the most efficient and correct code as the final output.

We conduct the ablation study using Qwen2.5-Coder-32B-Instruct and DeepSeek-V3 as LLM backbones, with the results presented in Table 2. The results show that removing any component significantly impacts both efficiency and correctness. Specifically, omitting Algorithmic Exploration in the Logic Domain (**Variant-1**) or Implementation Optimization in the Code Domain (**Variant-2**) leads to a marked decline in efficiency metrics across all three benchmarks. Additionally, removing Code Correctness Refinement (**Variant-3**) results in a significant drop in Pass@1. These results align with our expectations, as both Algorithmic Exploration and Implementation Optimization are designed for efficiency, while Code Correctness Refinement ensures the final code retains functional correctness after efficiency-driven steps.

5 Deeper Analysis

5.1 LLM4EFFI Uniqueness Analysis.

As mentioned in the Introduction, LLM4EFFI has two distinct features: **Uniqueness 1:** Separation of Efficiency Optimization into Logic and Code Domains, and **Uniqueness 2:** The Order of Correctness and Efficiency. To gain a deeper understanding of these unique advantages, we conducted the following comparative experiments:

- **w/o Uniqueness-1:** Rather than separating efficiency optimization into the logic and code domains, we prompt the LLM to generate code that is both efficient and correct. Then, based on the formalized task and the generated code, the LLM is queried to suggest any possible strategies for optimizing efficiency. Subsequently, we optimize the generated code according to these strategies, with the following steps remaining the same as in LLM4EFFI. It is important to note that

the difference between **w/o Uniqueness-1** and ECCO lies in the fact that ECCO provides correctness or efficiency strategies solely based on the code, whereas **w/o Uniqueness-1** generates efficiency-focused strategies based on both the formalized task and the generated code, followed by refinement for correctness.

- **w/o Uniqueness-2:** We changed the priority sequence of correctness and efficiency. In this approach, we first generate the code based on the formalized task and refine it for correctness. Then, we conduct algorithm exploration and implement optimal methods based on the formalized task and the refined code. Finally, we optimize the code using the explored algorithms and implementation suggestions.

The results of the uniqueness study are presented in Table 3. As shown, in the absence of **Uniqueness-1**, the 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 underscores the importance of separating efficiency optimization into the logic and code domains. This separation effectively breaks down the challenge of optimizing code efficiency into manageable steps, making the overall optimization process more focused and targeted. On the other hand, in the absence of **Uniqueness-2**, both efficiency and correctness saw significant declines across all benchmarks and different LLM backbones. The main reason for this is that optimizing correctness before efficiency limits the LLM's ability to explore efficient algorithms and practical optimizations. In fact, this approach often backfires, resulting in a situation where code correctness is sacrificed in the pursuit of efficiency. These findings further validate the "efficiency-first, correctness-later" strategy as a crucial approach for generating both efficient and correct code.

5.2 Performance of Different Difficulty Levels.

To evaluate the performance of LLM4EFFI on tasks of varying difficulty levels, we conducted an analysis across three levels—Easy, Medium, and Hard—using Mercury with DeepSeek-V3 as the backbone. The results are shown in Figure 3. LLM4EFFI consistently achieves the highest Beyond@1 metrics, outperforming other methods across tasks ranging from easy to difficult. This robust performance highlights the LLM4EFFI 's effectiveness in tackling a broad spectrum of challenges. Additionally, we observed a significant drop in performance for ECCO on hard-level tasks. This decline is mainly due to the difficulty of providing valid optimization suggestions for complex code, a challenge that remains substantial for LLMs.

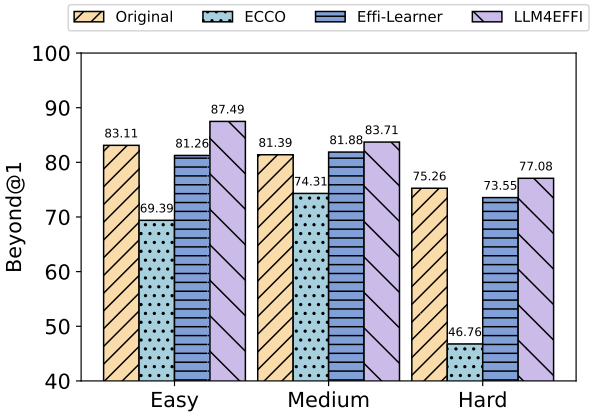

Figure 3: The Beyond@1 performance of LLM4EFFI on tasks of varying difficulty levels in Mercury, with DeepSeek-V3 as the backbone.

5.3 Case Study.

To provide a more intuitive demonstration of LLM4EFFI, we conducted a case study, with the process detailed in Appendix B. It can be observed that methods like ECCO and Effi-Learner, which generate code first and then optimize for efficiency, are constrained by the algorithmic design and overall structure of the initial code, leading to only incremental improvements. In contrast, LLM4EFFI breaks free from these constraints, enabling it to fully explore more efficient algorithms at a high level based on the task, while also incorporating practical level efficiency optimizations, thus achieving more effective efficiency optimization.

6 Conclusion

In this paper, we presented LLM4EFFI, a novel framework designed to generate both efficient and correct code. By separating efficiency optimization into the logic and code domains and adopting an "efficiency-first, correctness-later" approach, LLM4EFFI enables the exploration of a broader range of algorithmic solutions while maintaining functional correctness. Experimental results demonstrate LLM4EFFI 's robust performance, with consistent improvements in both efficiency and correctness across different LLM backbones.

Limitation

Although LLM4EFFI excels at generating both efficient and correct code, it also has some limitations. One major challenge is the trade-off between efficiency and maintainability. In some cases, the generated efficient code may become more complex and harder to read. Achieving the right balance between efficiency and maintainability is not always straightforward, and in certain cases, highly efficient code may sacrifice readability and ease of future modifications. Future work will focus on optimizing the LLM4EFFI to improve its scalability and extend its applicability to more complex software engineering tasks.

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.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. 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.

A Appendix of 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.

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 Process of LLM4EFFI.

As shown in Figure 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, 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 the 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 is then executed with these 20 test cases, recording the number of passed test cases ($Pass_{\text{t}} \leq 20$) and the number of successful executions for each test case ($Pass_{\text{c}} \leq 5$). 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 obtains a new batch of test cases and executes them again to gather new results. For the failed test cases, an iterative feedback mechanism is applied to optimize the code. Then, the code, enhanced with the iterative feedback, is executed once more, and the final passing results are recorded. All codes are then ranked in descending order based on their correctness, and the most accurate code is selected.

This process ensures the identification of the most optimal solution while maintaining both high efficiency and accuracy in code implementation.

B.2 Comparison of Methods.

In Figures 22 and Figures 23, we compare the code efficiency optimization processes of the three tools.

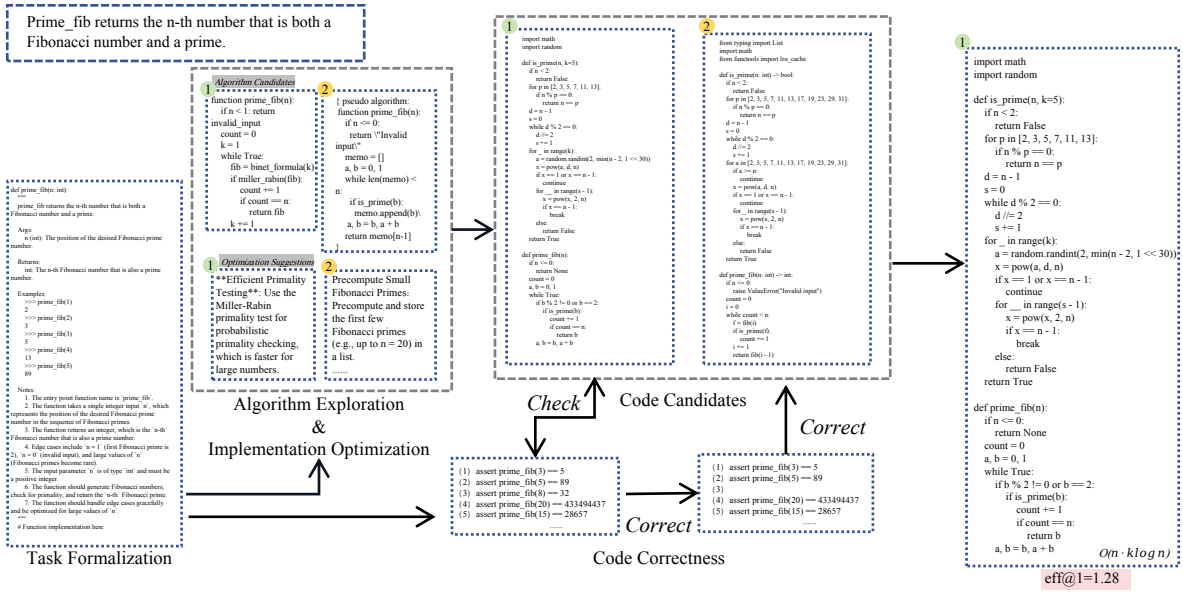

Figure 21: The figure illustrates the specific output of each subtask process of LLM4EFFI in solving algorithm problems.

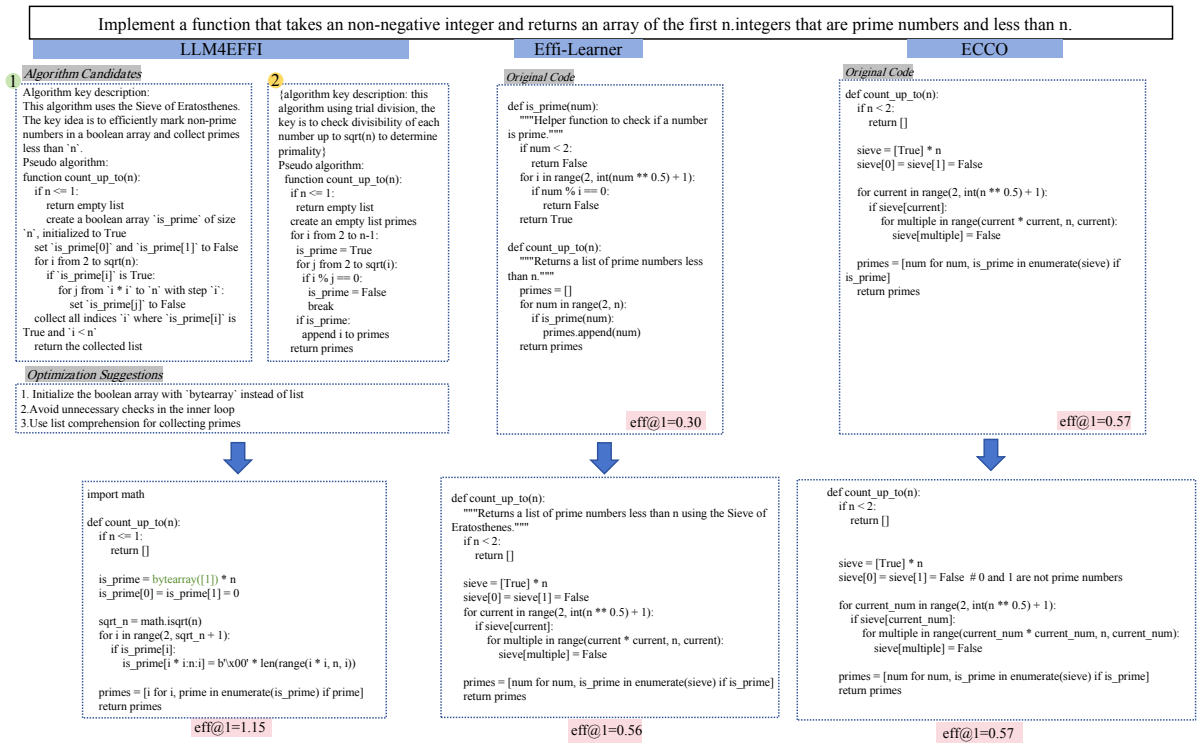

Figure 22: The diagram demonstrates how LLM4EFFI, Effi-Learner, and ECCO generate code. LLM4EFFI, through deep exploration of the algorithm domain, generates a set of efficient and high-quality algorithm candidates. However, the time complexity of these algorithms is similar, and there is no significant 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.

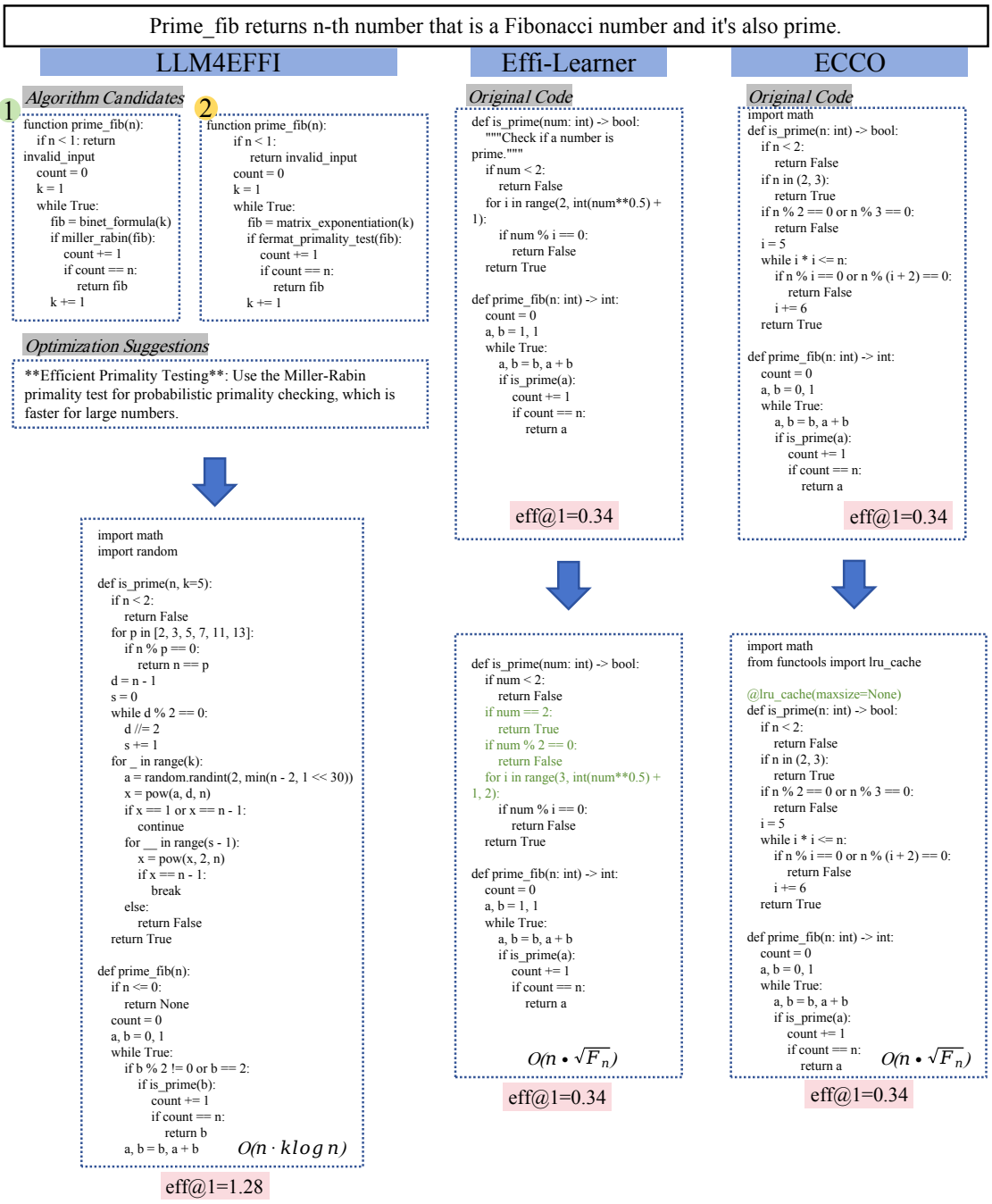

Figure 23: The figure illustrates the code generation process of LLM4EFFI, Effi-Learner, and ECCO. LLM4EFFI, through deep exploration of the algorithm domain, generates a set of efficient and high-quality algorithm candidates. By incorporating practical optimization suggestions, it ultimately produces 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.