ACECode: A Reinforcement Learning Framework for Aligning Code Efficiency and Correctness in Code Language Models

CHENGRAN YANG, Singapore Management University, Singapore
HONG JIN KANG, Singapore Management University, Singapore
JIEKE SHI, Singapore Management University, Singapore
DAVID LO, Singapore Management University, Singapore

CodeLLMs have demonstrated remarkable advancements in software engineering tasks. However, while these models can generate functionally correct code, they often produce code that is inefficient in terms of runtime. This inefficiency is particularly problematic in resource-constrained environments, impacting software performance and sustainability.

Existing approaches for optimizing code efficiency for CodeLLMs like SOAP and PIE exhibit certain limitations. SOAP requires a compatible execution environment and predefined test cases for iterative code modification, while PIE focuses on instruction tuning, improving efficiency but compromising correctness. These shortcomings highlight the need for a fine-tuning framework that optimizes both efficiency and correctness without relying on predefined test cases or specific execution environments.

To bridge this gap, we introduce ACECode, a reinforcement learning-based fine-tuning framework that aligns CodeLLMs with dual objectives of efficiency and correctness. ACECode combines three key steps: (1) generating code with an actor CodeLLM, (2) calculating a training-free reward signal derived from code execution feedback for each generated code, and (3) optimizing the CodeLLM via Proximal Policy Optimization (PPO) algorithm. This reward signal enables joint assessment of efficiency and correctness without manual labeling.

We evaluate ACECode by fine-tuning four SOTA (state-of-the-art) CodeLLMs and comparing their code with three baselines: original, instruction-tuned, and PIE-tuned CodeLLMs. Extensive experiment results suggest that ACECode significantly improves both the efficiency and correctness of generated code against all baselines for all CodeLLMs. Specifically, CodeLLMs fine-tuned with ACECode improve pass@1 by 1.84% to 14.51% and reduce runtime in 65% to 72% of cases compared to original CodeLLMs.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Open source model**.

Additional Key Words and Phrases: Code Language Models, Code Generation, Reinforcement Learning, Code Efficiency, Code Correctness, Software Engineering

ACM Reference Format:
Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. 2024. ACECode: A Reinforcement Learning Framework for Aligning Code Efficiency and Correctness in Code Language Models. 1, 1 (December 2024), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' Contact Information: Chengran Yang, cryang@smu.edu.sg, Singapore Management University, Singapore; Hong Jin Kang, hjkang@smu.edu.sg, Singapore Management University, Singapore; Jieke Shi, jiekeshi@smu.edu.sg, Singapore Management University, Singapore; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore.

ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Models (LLMs), such as GPT [5] and Qwen [6], have demonstrated remarkable advancements across various domains including software engineering [12, 15, 37]. Specialized LLMs for code like Code Alpaca [8], WizardCoder [26], Magicoder [40], and WaveCoder [42], known as CodeLLMs, are built on general-purpose LLMs and fine-tuned with code datasets spanning multiple programming languages. These CodeLLMs demonstrate exceptional performance across a range of code generation tasks, such as vulnerability patch generation [29] and code completion [14, 19].

While current CodeLLMs are capable of generating functionally correct code, this often comes at the expense of *code efficiency*. CodeLLM-generated code tends to be suboptimal in terms of runtime (i.e., the execution time for generated code), with prior studies [17, 18, 27, 33] showing that it can require 3-13× more runtime than human-written code.

The inefficiency of CodeLLM-generated code poses a significant challenge as CodeLLMs are increasingly integrated into real-world software development workflows. For instance, Google reports that CodeLLMs now generate over a quarter of new code in their products [20]. As such, inefficient code generated by CodeLLMs can significantly hinders the performance of software products and reduces their competitiveness, particularly in resource-constrained scenarios like mobile devices, cloud servers, and IoT systems. On the other hand, optimizing the efficiency of CodeLLM-generated code can contribute to environmental sustainability by enabling software products to consume less energy and resources, ultimately lowering their carbon footprint [33].

To date, few approaches have attempted to address this issue, including SOAP [17] and PIE [34]. SOAP [17] adopts a two-stage inference process, where CodeLLMs first generate code that is executed with test cases to collect runtime feedback. This feedback is then used to guide a second round of code generation aimed at reducing execution overhead. However, SOAP requires a compatible execution environment for LLM-generated code and pre-defined test cases for each task, which significantly restricts its out-of-the-box usability. Moreover, the two-stage inference process doubles the inference time and introduces additional overhead during code execution

On the other hand, PIE [34] crafts a dataset with efficient code snippets and applies instruction tuning for CodeLLMs to optimize code efficiency. While PIE optimizes code efficiency without relying on execution feedback, it applies instruction tuning with next-token prediction as its training objective. This approach primarily optimizes CodeLLMs as general language models, rather than addressing task-specific, multi-objective goals including code efficiency and correctness. Consequently, it leaves a gap in tailored fine-tuning methods specifically designed to optimize code efficiency. Moreover, a recent empirical study [35] highlights a crucial drawback of existing approaches including PIE: while code efficiency improves with these methods, they often do so at the expense of functional correctness (referred to as correctness). This drawback underscores the need for a fine-tuning framework specifically designed to optimize both code efficiency and correctness in CodeLLMs, eliminating the dependency on pre-defined test cases and a compatible execution environment during inference stage.

To address these limitations of existing approaches, we draw on techniques used for LLM alignment [28, 32, 39]—a methodology originally developed to tailor general LLM outputs to human preferences. LLM alignment often applies reinforcement learning from human feedback (RLHF) as a fine-tuning approach [28, 32, 39]. In RLHF, the LLM generates responses evaluated by a reward model (referred to as a "rewarder"), which estimates how much a human might favor the responses. The LLM's parameters are then iteratively updated to maximize the reward signals with reinforcement learning algorithms, progressively aligning the model with human preferences. In this work, we extend RLHF principles by tailoring specific LLM output, i.e., *generated code*, to meet fine-grained code quality requirements, i.e., *code efficiency and correctness.*

Applying RLHF to CodeLLMs to align the CodeLLM-generated code with correctness and efficiency requirements presents two primary challenges. First, RLHF typically requires a manually labeled dataset where LLM outputs are annotated with human preferences [10, 28, 32, 39] to train the reward model. Creating such a dataset is time-consuming and labor-intensive, requiring deep expertise in code algorithms and carrying the risk of subjective biases. The second challenge lies in representing both code efficiency and correctness within a rewarder, enabling the model to optimize for the dual objectives.

To address both challenges, we propose ACECode (**A**ligning Code **C**orrectness and **E**fficiency for **Code**LLMs), a reinforcement learning-based fine-tuning framework incorporating a training-free rewarder derived from code execution. ACECode eliminating the need for manually labeled datasets to train a reward model. Instead, it introduces a training-free, dual-objective rewarder that jointly evaluates code efficiency and correctness. Specifically, our rewarder assesses correctness based on the execution status of code against provided test cases and evaluates efficiency by comparing the runtime of generated code to that of a human-written reference solution. Moreover, to represent efficiency and correctness within one rewarder, we design the rewarder as a step function that penalizes incorrect or inefficient code while rewarding code that achieves both attributes, guiding the model to optimize toward both goals jointly. With this reward feedback, the CodeLLM is then optimized using Proximal Policy Optimization (PPO) [31] algorithm. To the best of our knowledge, this is the first RLHF method that improves both the efficiency and correctness of CodeLLM-generated code.

To evaluate the effectiveness of ACECode, we conduct a series of experiments using an extended version of EffiBench [18], a widely-used benchmark for code efficiency. Specifically, to enable a comprehensive evaluation of efficiency and correctness, we augment each coding task in EffiBench, originally paired with one ground-truth solution, with 10 additional human-written ground-truth solutions. We assess the performance of ACECode by fine-tuning four state-of-the-art open-source CodeLLMs and comparing their generated code with three baselines: original CodeLLMs, instruction-tuned CodeLLMs, and CodeLLMs fine-tuned with the state-of-the-art approach PIE [34]. Extensive experiment results suggest that ACECode significantly improves both the efficiency and correctness of generated code against all baselines for all CodeLLMs. Specifically, CodeLLMs fine-tuned with ACECode demonstrate improvements over the original CodeLLMs in code correctness ranging from 1.84% to 14.51% in terms of pass@1. Moreover, 65% to 72% of the code solutions generated by CodeLLMs fine-tuned with ACECode exhibit reduced runtime compared to those produced by the original CodeLLMs. Furthermore, ACECode outperforms instruction tuning and PIE in terms of code correctness and efficiency, achieving improvements of up to 51.15% in pass@1 and 23.18% in average execution time compared to instruction tuning, and up to 14.41% in pass@1 and 11.45% in average execution time compared to PIE.

We summarize our contributions in this paper as follows:

- **The first RLHF framework for optimizing dual attributes of LLM-generated code**: We introduce ACECode, the first reinforcement learning-based fine-tuning framework for optimizing code efficiency and correctness.
- **Proposing a training-free rewarder for assessing code correctness and efficiency**: ACECode provides a novel rewarder that jointly assesses code efficiency and correctness, derived directly from code execution feedback without requiring a manually labeled dataset.
- **Extending Existing Benchmark:** We extend existing benchmark EffiBench [18] by adding 10 additional human-written ground-truth solutions for each task, making the dataset more robust for evaluating efficiency and correctness improvements.

- **Demonstrating the effectiveness of ACECode**: We evaluate ACECode's effectiveness on the EffiBench benchmark with multiple state-of-the-art open-source CodeLLMs. Results show that ACECode significantly enhances code efficiency and correctness, achieving improvements of up to 14.51% in pass@1 rate and up to 10.86% in average execution time against vanilla CodeLLMs.

The remaining part of this paper is organized as follows. Section 2 provides an overview of background concepts, related work, and formalizes the problem definition. Section 3 introduces ACECode, detailing its architecture and reward design. Section 4 presents the experimental setup, evaluation metrics, and implementation details. Section 5 discusses the results of our work. Section 6 discusses broader implications, multi-objective optimization, and threats to validity. Finally, Section 7 concludes the paper and outlines future work.

2 Background and Related Work

2.1 Related Work

2.1.1 LLMs for Code. Recent research has focused on improving the capacity of LLMs to perform code-related tasks. Gnerally, two lines of research have been proposed to enhance the performance of LLMs in code Generation: (1) pre-training LLM on large-scale code corpora, like AlphaCode [23], StarCoder [25], and CodeT5 [38], and (2) fine-tuning foundamental general-purpose LLMs on code-related tasks or with code-related instruction datasets, like WizardCoder [26], Magicoder [40], and WaveCoder [42]. Those LLMs specifically optimized for code-related tasks are referred to as CodeLLMs. However, existing empirical studies [18, 35] show that those CodeLLMs often overlook optimizing generated code's execution efficiency, resulting in 3 to 13× slower code than human-written peers.

Limited works including SOAP [17] and PIE [34] have attempted to optimize the execution efficiency of CodeLLM-generated code. SOAP [17] leverages in-context learning with multiple iterations to enhance CodeLLM efficiency during inference. Specifically, SOAP operates within a code execution environment and iteratively prompts the CodeLLM to refine its generated code by providing execution feedback after each iteration. However, SOAP requires a compatible execution environment and introduces additional overhead to model generation, which hinders its out-of-box usage. PIE [34] improves code efficiency on C++ by creating a dataset of efficient code snippets and applying instruction-based tuning [35], though it sacrifices code correctness. In contrast, CodeLLMs fine-tuned by ACECode are free from execution environment dependencies, incur no additional overhead during inference, and demonstrate improvements in both code correctness and efficiency, overcoming the limitations of previous approaches [17, 34].

2.1.2 Evaluating CodeLLM for Code Generation. Code generation, which involves generating code snippets or functions based on natural language prompts or docstrings, has received increasing attention in recent years [23, 40, 42]. To evaluate LLM performance on this task, various benchmarks have been proposed, primarily focusing on the correctness of generated code across different scenarios. For example, HumanEval assesses functional correctness by executing generated code on a set of test cases [9]. DS1000 evaluates code generation capabilities with a focus on third-party libraries within the data science domain [21]. EvoCodeBench measures code generation performance at the repository level [22]. Recently, EffiBench [18] and ECCO [35] are introduced to evaluate the efficiency of generated code. Both works release a code efficiency benchmark by crafting coding tasks, reference solutions on Python, and corresponding test cases to evaluate the performance of CodeLLMs. Findings from EffiBench [18] indicate that CodeLLMs all produce code that is suboptimal in execution runtime, while ECCO [35] points out that existing works on improving the efficiency of CodeLLMs-generated code sacrifices their functional correctness, underscoring the need to consider both code efficiency and correctness in optimizing CodeLLMs.

2.2 RLHF

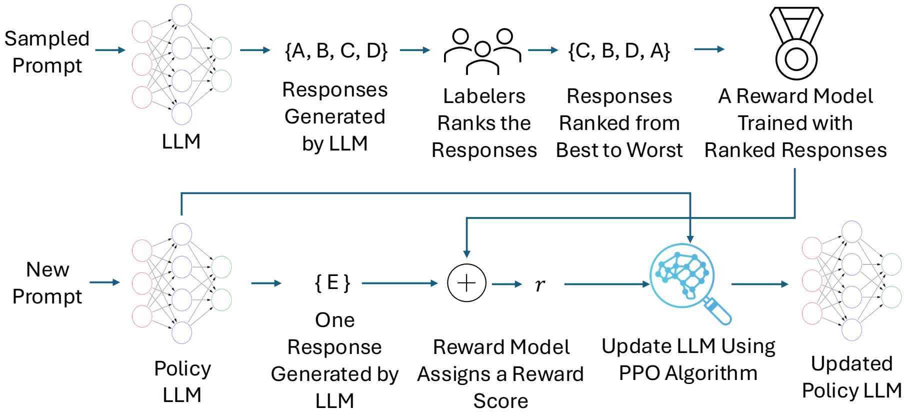
Sampled Prompt
LLM
{A, B, C, D}
Responses Generated by LLM
Labelers Ranks the Responses
{C, B, D, A}
Responses Ranked from Best to Worst
A Reward Model Trained with Ranked Responses
New Prompt
Policy LLM
{ E }
One Response Generated by LLM
Reward Model Assigns a Reward Score
r
Update LLM Using PPO Algorithm
Updated Policy LLM

Fig. 1. Overview of RLHF. Generally, the process begins with the LLM generating multiple responses for a given prompt, which are then ranked by human labelers based on their preference. These ranked responses are used to train a reward model that assigns a reward score to evaluate the human preference of newly generated responses. The LLM is then fine-tuned using the Proximal Policy Optimization (PPO) algorithm to optimize its outputs for higher reward scores, iteratively aligning LLM's outputs with human preferences.

Reinforcement learning from human feedback (RLHF) is a widely-used fine-tuning technique for aligning LLM outputs with human preferences [10, 28, 32, 39]. We introduce the overview of RLHF in Figure 1. During the pre-processing steps of RLHF, human annotators label pairwise LLM outputs as a pair of preferred and non-preferred responses, which is then used to train a rewarder that quantitatively estimates human preferences. During RLHF fine-tuning, the target LLM generates responses for the prompts in the training dataset. The rewarder receives the generated responses and assigns a scalar reward score approximating the human preferences towards the responses given the prompt. The target LLM is then iteratively fine-tuned using a reinforcement learning algorithm, typically PPO [31], to optimize its parameters by maximizing the reward. RLHF has proven effective in guiding LLMs to generate trustworthy, secure, and non-toxic text [10, 28]. Unlike previous RLHF frameworks designed for general-purpose tasks and require manually labeled datasets to train a rewarder, ACECode specifically targets code correctness and efficiency and proposes a training-free and dual-objective rewarder grounded in code execution feedback.

2.3 Task Definition

We define the dual-objective code generation task as generating code solutions $\hat{C}$ that are both efficient (G_e) and correct (G_c) for a natural language instruction I, as a sequence-to-sequence problem. This task is modeled as a sequence-to-sequence problem, where the input is a natural language instruction I and the output is code snippets $\hat{C} = (\hat{c}_1, ..., \hat{c}_t)$ that satisfies the dual objectives of G_e and G_c.

To achieve this, we optimize the model parameters θ to maximize the conditional probability of generating a correct and efficient code sequence, incorporating the dual-objective reward $R(G_e, G_c)$:

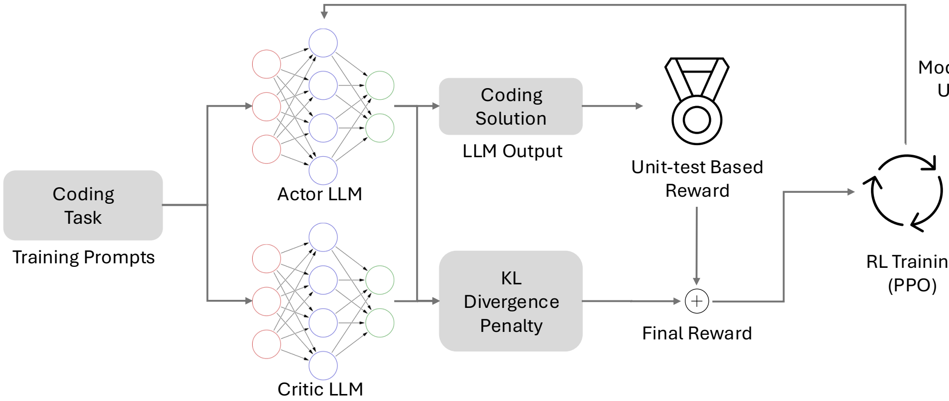

Fig. 2. Overview of ACECode with PPO training. The Actor LLM, which serves as the target model for optimization, takes prompts as input and generates code snippets at each step. For each LLM output, the Unit-test Based Rewarder executes the generated code with all test cases, assigning a reward score to the generated code, representing both code correctness and efficiency based on execution feedback. This reward score is used to guide the Actor LLM in optimizing its policy via the PPO algorithm. Meanwhile, the Critic LLM estimates the value function for each Actor LLM output, facilitating more stable policy optimization.

$$\max_{\theta} \mathbb{E}_{\hat{C} \sim p_\theta(\cdot|I)} \left[R(G_e(\hat{C}), G_c(\hat{C})) \right] \tag{1}$$

where $R(G_e(\hat{C}), G_c(\hat{C}))$ is a reward function explicitly quantifying code efficiency and correctness and $p_\theta(\cdot|I)$ represents the model's policy for generating code sequences. The CodeLLM is fine-tuned to adjust its policy to maximize the expected reward, aligning its outputs with the dual objectives of efficiency and correctness.

3 Methodology

We propose ACECode, a novel reinforcement learning-based fine-tuning framework with dual objectives: code correctness and efficiency. ACECode consists of three main components: (1) code generation, (2) reward calculation, and (3) policy optimization through PPO algorithm. Figure 2 illustrates the ACECode architecture, comprising an actor LLM (π_θ) responsible for generating code snippets, a rewarded $\mathcal{R}$ to output reward signals, and a critic LLM (V_π) to estimate value functions for stable policy updates. Both LLMs are optimized using the PPO algorithm [31], guided by a reward signal $\mathcal{R}$ that estimates code quality based on efficiency and correctness. Specifically, the actor LLM serves as the target model for optimization, responsible for producing code solutions based on natural language prompts. Meanwhile, the Critic LLM serves as a stability mechanism in the training process, estimating the value function for each code snippet generated by the Actor LLM. The value function quantifies the expected cumulative reward of each code snippet, which stabilizes the policy updating for the actor LLM. Finally, the tuned actor LLM (π_θ) is extracted as the optimized target model for generating efficient and correct code solutions.

3.1 Code Generation

The first step in ACECode's framework is code generation. For each coding task I, the actor LLM (π_θ) generates a code snippet $\hat{C}$. We apply prompts in the same format as prompts in existing benchmark of code efficiency, i.e., EffiBench [18]. We show the prompt template in Table 1. Besides, to align the

Table 1. Prompt templates for LLMs.

Prompt Template For Code Generation
Instruction: Please based on the task description write Python Solution to pass the provided test cases. *You must follow the following rules:* *The code should be in* ”’python\n[Code]\n”’ *block.* *Second, You should not add the provided test cases into the code.* *Third, You do not need to write the test cases, we will provide the test cases for you.* *Fourth, Only include the Python solution, without any additional explanation.* *Fifthly, import only the necessary modules; avoid using import *.* *Finally, You should make sure that the provided test cases can pass your solution.* *### Here is a example:* *{Example Instruction}* *{Example Code Solution}* *### Task Description:* *{Task Description}*

output format of the LLM with the task requirements, we add an example pair $(I_{example}, C_{example})$ into the prompt for in-context learning. In-context learning has proven to be effective for aligning output format [7].

The actor LLM generates N responses for each input prompt. We then extract the code snippets $(\hat{C}_1, ..., \hat{C}_N)$ from each response by matching the format of the coding block that is explicitly defined in the instruction. Notably, following standard practice in RLHF [10, 28], the temperature of LLM in the fine-tuning stage is set to 0.85 to generate diverse responses. $(C, \hat{C}_1, ..., \hat{C}_N)$ is then passed to the rewarder that will assess and produce scores representing code efficiency and correctness jointly.

3.2 Rewarder

The rewarder $\mathcal{R}$ serves as a proxy for code efficiency and correctness assessment, providing a scalar reward signal for each input $\hat{C}$. The reward signal then guides the optimization of the actor LLM (π_θ) with dual objectives, i.e., generating efficient and correct code, through the PPO algorithm. Different with common rewarders that are trained with human-labeled dataset [10, 28] and are prone to subjective bias, ACECode's $\mathcal{R}$ applies a training-free and objective evaluator grounded by code execution, eliminating the need for manual labeling and reducing the risk of subjective bias.

Designing $\mathcal{R}$ to represent both code efficiency and correctness presents several challenges:

- **C1. Requiring human-labeled dataset**: Rewarders often require a human-labeled dataset for training [10, 28], which is prone to subjective bias and requires substantial labeling effort.
- **C2. Representing code efficiency and correctness in one rewarder**: The rewarder must encapsulate both objectives including code correctness and efficiency; otherwise, the actor LLM may prioritize one objective over the other during fine-tuning.
- **C3. Inaccurate runtime measurements**: Some code snippets with short execution time can have high variance in the measurement of execution runtime due to potential concurrent processes and can be inaccurate if measured with profiling tools [4].

The following subsections detail our approach to address these challenges.

3.2.1 Objective Evaluator for Efficiency and Correctness. To address challenge C1, we do not manually label code snippets with efficiency and correctness scores to train a rewarder. Instead, we use an objective and training-free evaluator based on code execution that measures code efficiency and correctness directly as the rewarder. Specifically, for a given pair $\left(C, \hat{C}\right)$, we obtain a set of ground-truth test cases $T = \{t_1, t_2, ..., t_n\}$ and execute both C and $\hat{C}$ on T. We record the compiler signals, which indicate whether the code encounters a compilation error, and execution runtimes $\left(E_C, E_{\hat{C}}\right)$ for the ground-truth and generated solutions, respectively. We then apply a customized reward function $\mathcal{R}$ to calculate the reward score that reflects both code efficiency and correctness based on the compiler signals and runtime values.

3.2.2 Reward Function. To address challenge C2, we design $\mathcal{R}$ as a step function that evaluates the generated code's correctness and efficiency based on compiler outcomes and runtime performance:

$$\mathcal{R}(\hat{C}, C) = \begin{cases} 0.5 + 0.5 \cdot \min\left(\left(\frac{E_C^t}{E_{\hat{C}}^t}\right)^k, 1\right), & \text{if } \hat{C} \text{ passes all unit tests} \\ -0.3, & \text{if } \hat{C} \text{ fails any unit test} \\ -0.5, & \text{if } \hat{C} \text{ encounters a compile error} \end{cases} \tag{2}$$

where k is a penalty factor controlling the sensitivity to efficiency gain.

This reward structure encourages π_θ to generate code that is both correct and efficient. Following existing approaches [10, 28], we normalize the reward values from -1 to 1 to stabilize the training process. Firstly, if $\hat{C}$ fails to compile or pass any unit test, it receives a negative reward. Code that fails to compile receives a penalty of -0.5, set as half the worst-case value to avoid making the model overly cautious and under-optimizing. A smaller penalty of -0.3 is assigned for code that fails any unit test, as compilation correctness is a prerequisite for functional correctness. Conversely, if $\hat{C}$ passes all tests, it receives a positive reward comprising a fixed score 0.5 representing code correctness and an adaptive score $0.5 \cdot \min\left(\left(\frac{E_C^t}{E_{\hat{C}}^t}\right)^k, 1\right)$ representing code efficiency. To encourage balanced optimization of both objectives, the fixed score for correctness is set to 0.5 and the adaptive score for efficiency is capped at a maximum of 0.5. Concretely, the code efficiency of $\hat{C}$ is quantified as the ratio of its runtime $E_{\hat{C}}^t$ to that of the ground-truth E_C^t.

3.2.3 Precise Runtime Measurement. Existing benchmark EffiBench [18] applies profiling tool `memory-profiler`[1] to execute code once and extract the runtime. However, as mentioned by official Python documentation[2], profiling tools would introduce additional overhead to the runtime measurement of Python code and are not designed for benchmarking purposes. Meanwhile, we observe that the runtime of each code may be affected by concurrent processes, resulting in inconsistent runtime for each execution.

Therefore, to address challenge C3, following recommendations from the official Python documentation[3], we use Python's `timeit` module [4]. This allows reasonable assessments of execution runtime with less overhead. Additionally, to mitigate the effect of randomness in the runtime between different executions, we implement a repeated execution strategy. We calculate the runtime value by executing each generated code snippet n times and computing the average, $\widetilde{E}_{\hat{C}^t}$. We define two parameters for repeating execution:

[1] https://pypi.org/project/memory-profiler/

[2] https://docs.python.org/3.9/library/profile.html

[3] https://docs.python.org/3.9/library/profile.html

- n_{min}: the minimum number of executions needed to stabilize runtime measurements.
- t_{max}: the maximum allowable accumulated runtime to avoid an infinite loop.

The total number of executions (n) is then calculated as:

$$n = \max\left(n_{min}, \frac{t_{max}}{n_{t_{max}}}\right) \tag{3}$$

where $n_{t_{max}}$ is the number of completed executions when the accumulated runtime reaches t_{max}.

We also consider the overhead of importing libraries. We move all import statements to the `setup` parameter of `timeit`, ensuring they are executed only once and excluded from measurement cycles.

Finally, for all input code $(\hat{C}_1, ..., \hat{C}_N)$, the rewarder assigns reward values, which are then used to guide the PPO training.

3.3 PPO Training Framework

We use the PPO algorithm [31] to fine-tune the actor LLM π_θ with the reward signal R. This code generation process is formulated as a sequential discrete Markov Decision Process, where each step in the generation process is defined by tuple $(State, Action, Policy, Reward)$ and solved by PPO. At each time step t, representing the token length of the generated code so far, the components of the MDP process are defined as follows:

- **State** S_t: The state at time step t consists of the input prompt I and the partially generated code snippet $\hat{C}_1 = (\hat{c}1, \ldots, \hat{c}t - 1)$. This concatenation forms the input context up to token t.
- **Action** A_t: The action at each time step t is defined as generating the next token $\hat{c}_t$ in the code sequence.
- **Policy** π_θ: The policy is represented by actor LLM π_θ, which performs the action A_t, i.e., generates the next token $\hat{c}_t$ given the state S_t.
- **Rewarder** $\mathcal{R}_t$: The rewarder $\mathcal{R}_t$ is applied at the end of each generation episode, i.e., when the complete code snippet $\hat{C}$ is produced. The rewarder produces a scalar reward value that reflects both code correctness and efficiency, as detailed in Section 3.2.
- **Advantage** A_t: The advantage A_t quantifies the value of the action A_t relative to the critic LLM's estimated value function V_π, given the current state S_t. It is computed using the critic model V_π and the reward signal $\mathcal{R}_t$. Specifically, for each time step t, we calculate A_t as follows:

$$A_t = \mathcal{R}_t + \gamma V(s_{t+1}) - V(s_t) \tag{4}$$

where $V(s_t)$ is the value of the current state as estimated by the critic LLM and γ is a discount factor.

The objective of training is to maximize the expected reward by optimizing the policy π_θ. This objective can be formalized as:

$$\max_\theta \ \mathbb{E}_{I \sim \mathcal{I},\ \hat{c} \sim \pi_\theta(\cdot|I)} \left[\mathcal{R}(\hat{c}, I, c)\right] \tag{5}$$

where $\mathcal{I}$ is the dataset of input prompts, and c is the ground-truth code for each prompt. Specifically, the PPO loss function $L_{\text{PPO}}(\theta)$ is defined as:

$$L_{\text{PPO}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)A_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)A_t\right)\right] \tag{6}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the current and previous policies for action a_t. $\text{clip}(\cdot, 1-\epsilon, 1+\epsilon)$ ensures the ratio is constrained within a small range to prevent large updates, with ϵ being a hyperparameter. Concretely, the clip function is defined as follows:

$$\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) = \begin{cases} 1-\epsilon, & \text{if } r_t(\theta) < 1-\epsilon, \\ r_t(\theta), & \text{if } 1-\epsilon \le r_t(\theta) \le 1+\epsilon, \\ 1+\epsilon, & \text{if } r_t(\theta) > 1+\epsilon. \end{cases} \tag{7}$$

In parallel, the critic LLM V_π is updated by minimizing the following value loss:

$$L_{\text{value}}(\pi) = \mathbb{E}\left[(V_\pi(s_t) - (R_t + \gamma V_\pi(s_{t+1})))^2\right] \tag{8}$$

where $V_\pi(s_t)$ is the value estimate for the current state; $\mathcal{R}t + \gamma V\pi(s_{t+1})$ serves as the target value.

To prevent overfitting during policy updates, we incorporate a per-token KL divergence penalty into the PPO objective, which is demonstrated to be effective in prior work [28]. This adjustment controls how closely the updated policy adheres to the previous policy, $\pi_{\theta_{\text{old}}}$, and the combined loss for PPO is expressed as:

$$L_{\text{total}} = L_{\text{PPO}}(\theta) + \lambda L_{\text{value}}(\phi) - \beta \text{KL}(\pi_\theta \parallel \pi_{\theta_{\text{old}}}) \tag{9}$$

where λ is a weighting coefficient for the value loss and β is the coefficient for the KL penalty, which mitigates excessive divergence between the current and old policies.

4 Experimental Setup

4.1 Dataset

To evaluate ACECode, we use an extended version of the EffiBench [18] dataset, which is designed for assessing code efficiency in CodeLLMs. EffiBench consists of 1,000 efficiency-focused coding tasks extracted from LeetCode [2], each paired with a single ground-truth solution.

To compute the reward for a CodeLLM-generated code solution, ACECode determines the level of code efficiency by comparing its execution runtime against the ground-truth reference solution, which ideally serves as an upper bound for runtime efficiency. While EffiBench extracts one most-upvoted solution from LeetCode's discussion forum as a reference solution for each coding task, we observe that these solutions are not always the most runtime-efficient, as some prioritize readability or memory usage over runtime efficiency. To address this, we extend EffiBench by gathering at most 15 most-upvoted solutions for each coding task, executing them, and then picking the solution with the shortest runtime as a new reference solution. This approach establishes an accurate benchmark for the upper bound efficiency of code solutions for each coding task, allowing for a more precise calculation of code efficiency reward and fair evaluation. Note that coding tasks that are restricted to premium users or contain faulty test cases are excluded from our dataset. The resulting extended dataset, which we refer to as EffiBench+, comprises 797 coding tasks and 8,520 code solutions. We randomly split EffiBench+ into training and test sets with an 8:2 ratio. This test set aligns in size with HumanEval [9], one of the most widely used benchmarks for code generation.

4.2 Selecting CodeLLMs

We fine-tune four state-of-the-art (SOTA) open-source CodeLLMs selected from the top-performing models on the Evalplus [24] leaderboard. Specifically, we choose SOTA models including Magicoder [40], DeepSeekCoder [43], CodeQwen1.5 [6], and WaveCoder [42]. Specifically, we select the model version with parameters less than 10B due to the computational constraints of our training environment. We give the details of each selected CodeLLM in Table 2. Note that as ACECode requires updating the internal states of CodeLLMs via reinforcement learning, we are unable to evaluate commercial closed-source models.

Table 2. Summary of Selected CodeLLMs

Model Name	Number of Parameters	Institution
DeepseekCoder-6.7B-instruct	6.7B	DeepSeek AI
Magicoder-S-DS-6.7B	6.7B	Intelligent Software Engineering-UIUC
CodeQwen1.5-7B-Chat	7B	Alibaba Cloud
WaveCoder-Ultra-6.7B	6.7B	Microsoft

Magicoder [40] is a family of CodeLLMs designed for code generation tasks. These models are trained using OSS-Instruct, a novel dataset that leverages open-source code snippets to generate diverse and realistic instruction data. This methodology aims to mitigate inherent biases present in LLM-synthesized data by incorporating a wealth of open-source references, resulting in more diverse, realistic, and controllable outputs.

DeepSeekCoder [43] is an open-source series of CodeLLMs that are designed to enhance code generation and understanding tasks. The base model of DeepSeekCoder is trained from scratch on a massive dataset of 2 trillion tokens, comprising 87% code and 13% natural language in both English and Chinese. They utilize a 16K context window and a fill-in-the-blank task to support project-level code completion and infilling tasks.

CodeQwen1.5 [6] is a code-specific variant of the Qwen1.5 LLM, which is pre-trained on approximately 3 trillion tokens of code-related data, encompassing 92 programming languages. The model supports a context length of up to 64,000 tokens.

WaveCoder [42] is a series of CodeLLMs fine-tuned on an instruction dataset called CodeSeaX-Dataset, which comprises 19,915 instruction instances across four tasks: code generation, code summarization, code translation, and code repair. The fine-tuning process employs a generator-discriminator framework to ensure high-quality and diverse instruction data.

4.3 Implementation Details

We implement ACECode based on the HuggingFace Transformers platform [1] and OpenRLHF framework [16].

To ensure consistent reward calculation, we execute each ground-truth and CodeLLM-generated solution pair consecutively on the same machine, reducing potential variance introduced by the execution environment. Meanwhile, we observe that more than 95% ground-truth solutions in EffiBench require less than 1 seconds to execute in our local environment, so we set the maximum allowable accumulated execution time (t_{max}) in equation 3 to 3 seconds. We also set the execution count to 10 to ensure stable evaluation results. Furthermore, any code execution exceeding 10-seconds timeout would be terminated to avoid infinite loops. To facilitate reproducibility, we set the generation parameters with a temperature of 0 and top-p of 1 during the inference stage, ensuring deterministic outputs from the CodeLLMs for identical prompts.

During the RLHF training phase, we separate the parameters of actor and critic networks. As recommended in OpenRLHF [16], we set the training batch size as 64, rollout batch size as 512, maximum generate length as 1024, learning rate for actor model as $5e^{-7}$, learning rate for critic model as $9e^{-6}$, and KL penalty coefficient as 0.01. Following standard practice in RLHF [16, 41], we configure the sampling strategy of LLM by setting the temperature and top-p as 1, and we sample 5 responses for each prompt in training. To accelerate training, we apply FlashAttention [11] and

DeepSpeed [30]. We randomly sample 5% of the training data as a validation set and select the checkpoint with the best validation performance for final evaluation.

4.4 Baselines

We consider three baseline settings for evaluating the effectiveness of ACECode:

- **Original CodeLLMs**: We evaluate the correctness and efficiency of the code from CodeLLMs fine-tuned with ACECode against the original CodeLLMs without fine-tuning.
- **CodeLLMs with Instruction Tuning**: We perform instruction tuning on the original CodeLLMs using the same training split from EffiBench as ACECode and compare their performance with CodeLLMs fine-tuned by ACECode. The instruction is constructed with the task description, a one-shot example of task description and code, and the ground-truth solution. The template of the task description is the same as ACECode, which is illustrated in Table 1. We fine-tune CodeLLMs using the instructions with the next token prediction objective.
- **PIE [34]**: We consider PIE as a strong baseline for code efficiency optimization. PIE applies instruction tuning with performance-conditioned generation to optimize the efficiency of CodeLLM-generated code. Performance-conditioned generation involves tagging code solutions with performance scores during training (e.g., the top 10% most efficient solutions are labeled as "10/10," the next 10% as "9/10," and so on). PIE's dataset consists of pairs of efficient and inefficient code snippets for each task, written in C++. During the training stage, the ground-truth performance scores are provided for each input code to quantify the extent of code efficiency. During the inference stage, PIE prompts the LLM to generate a code solution with a 10/10 performance score.
  As the released model checkpoints of PIE are optimized for C++, and our training dataset is focused on Python, we extend PIE approach to Python for a fair comparison. Specifically, we follow PIE's instruction-tuning methodology while leveraging our extended version of EffiBench (see Section 4.1). Our extension provides multiple ground-truth code solutions per coding task, which are executed once before training to assign performance tags.

4.5 Evaluation Metrics

4.5.1 Code Correctness. we use the pass@1 metric to evaluate the code correctness of CodeLLMs. The pass@1 metric measures the proportion of generated code solutions that pass all test cases on the first attempt. Pass@1 ranges from 0 to 1, with higher values indicating better code correctness:

$$\text{pass@1} = \frac{\text{Number of passed solutions}}{\text{Total number of coding tasks}} \tag{10}$$

4.5.2 Code Efficiency. Existing benchmarks for evaluating code efficiency, such as EffiBench [18], use metrics like Average Execution Time (AET) and Normalized Average Execution Time (NAET). AET calculates the arithmetic mean execution time of generated code solutions, while NAET measures the relative arithmetic mean execution time of generated solutions compared to the ground-truth solution. However, we empirically find that AET and NAET are disproportionately influenced by outliers because they compute the arithmetic mean. For instance, if a generated code solution for a particular task has an exceptionally longer execution time than other tasks, it may dominate the AET, skewing the metric. Similarly, if one solution's inefficiency relative to the ground-truth solution is significantly higher than other tasks, NAET would also become biased toward this outlier.

To mitigate this bias, we propose a new metric called Efficiency Code Counting (ECC), inspired by differential testing [13]. ECC calculates the percentage of code solutions generated by CodeLLMs fine-tuned with ACECode that exhibit lower execution times than those generated by the original CodeLLM for the same tasks and test cases. Formally, ECC is defined as:

$$\mathrm{ECC} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left(t_i^{\mathrm{originLLM}} > t_i\right) \tag{11}$$

where N is the number of code generated by CodeLLM, t_i is i-th code task, and $t_i^{\mathrm{originLLM}}$ is the accumulated execution time of the original CodeLLM generated solutions. The ECC metric ranges from 0 to 1 and is not affected by extreme outliers. An ECC value above 0.5 indicates that more than half of the solutions generated by the fine-tuned CodeLLM are more efficient than those of the original model.

One limitation of ECC is that it only counts the number of more efficient solutions generated by the fine-tuned CodeLLM without accounting for the magnitude of the efficiency improvement. To address this, we introduce one additional metric: Geometric Execution Time (GET), inspired by recent works on accelerating software testing [36]. Unlike AET and NAET which apply arithmetic means of execution time, GET calculates the geometric mean of execution time. The geometric mean allows us to capture the average improvement in efficiency without undue influence from outlier values. Specifically, we define GET as follows:

$$GET = \sqrt[N]{\prod_{i=1}^{N} \frac{t_i}{n_i}} \tag{12}$$

where N is the number of code tasks, t_i is the accumulated execution time of the code solution for i-th task, and n_i is the number of code executions through adaptive repeated execution strategy. GET calculates the geometric mean of execution time for all code tasks. A lower GET value of the fine-tuned CodeLLM relative to the original CodeLLM indicates better runtime efficiency of the generated code.

To ensure fair and consistent comparison with EffiBench, we evaluate code efficiency only on tasks for which both the fine-tuned and original CodeLLM generate correct solutions. In this paper, we consider both ECC and GET as the primary metrics for assessing code efficiency improvements.

5 Experimental Results

This section describes the evaluation results of CodeLLM fine-tuned by ACECode compared to the original CodeLLM. Specifically, our evaluation focuses on the following two research questions (RQ):

- **RQ1:** How do the CodeLLMs fine-tuned by ACECode perform in terms of code correctness and efficiency?
- **RQ2:** How do the parameters of ACECode affect the performance of the fine-tuned CodeLLMs?

5.1 RQ1: Code Correctness and Efficiency of CodeLLMs

In this RQ, we fine-tune CodeLLMs by ACECode and evaluate the correctness and efficiency of the generated code. We compare the performance of these models with the original CodeLLMs, instruction-tuned CodeLLMs, and CodeLLMs fine-tuned using PIE [34].

5.1.1 Comparing With Original CodeLLMs. We show the evaluation results of CodeLLMs fine-tuned by ACECode against the original CodeLLMs in Table 3. We observe that all CodeLLMs

Table 3. Correctness and Efficiency of Code Generated by Original CodeLLMs and CodeLLMs Fine-tuned by ACECode.

	ECC (%)	GET ↓	Pass@1 (%) ↑
CodeQwen1.5-7B-Chat	N/A	1.121×10^{-3}	41.94
+ ACECode	72.3%	1.065×10^{-3} (↓4.98%)	46.77 (↑11.52%)
DeepseekCoder-6.7B-instruct	N/A	8.551×10^{-4}	44.44
+ ACECode	61.2%	8.505×10^{-4} (↓0.52%)	48.41 (↑8.93%)
WaveCoder-Ultra-6.7B	N/A	1.749×10^{-3}	42.86
+ ACECode	65.4%	1.559×10^{-3} (↓10.86%)	43.65 (↑1.84%)
Magicoder-S-DS-6.7B	N/A	1.152×10^{-3}	49.21
+ ACECode	67.9%	1.063×10^{-3} (↓7.69%)	56.35 (↑14.51%)

As defined in Section 4.5, ECC measures the percentage of code generated by tuned CodeLLMs that has less execution time than that of original CodeLLMs. GET measures the geometric average execution time of the generated code. Pass@1 measures the percentage of generated code that passes the test cases at the first attempt.

fine-tuned by ACECode outperform the original CodeLLMs in terms of both code correctness and efficiency. Specifically, the ECC score of all CodeLLMs fine-tuned by ACECode is higher than 60%, indicating that at least 60% of the code generated by all CodeLLMs fine-tuned by ACECode has executes in a shorter time than the original CodeLLMs. Similarly, GET for all CodeLLMs fine-tuned by ACECode decreased from 0.52% to 10.86% compared to the original CodeLLMs. Both GET and ECC scores illustrate the superior efficiency of the code generated by CodeLLMs fine-tuned by ACECode. Moreover, we can also observe that the pass@1 score of all CodeLLMs fine-tuned by ACECode is higher than the original CodeLLMs by from 1.84% to 14.51%, showing the improvement in code correctness of the generated code.

Besides, we observe that the less efficient the original CodeLLMs, the more significant the efficiency improvements achieved by ACECode. For all CodeLLMs in this study, the higher the original CodeLLM's GET score, the more substantial the efficiency gains achieved by ACECode. We attribute this trend to an upper bound on achievable efficiency improvements. CodeLLMs that are less efficient offer more room for targeted fine-tuning improvements.

5.1.2 Comparing With Instruction Tuned CodeLLMs. Table 4 presents the evaluation results comparing CodeLLMs fine-tuned with ACECode to those trained via instruction tuning. CodeLLMs fine-tuned by ACECode consistently outperform their instruction-tuned counterparts in both code correctness and efficiency.

Specifically, ACECode achieves substantial improvements in code correctness from 5.79% to 51.15% in terms of pass@1. Similarly, the runtime efficiency of ACECode-fine-tuned CodeLLMs surpasses instruction-tuned models by 9.92% to 23.18% in terms of GET score, highlighting the superior efficiency of ACECode in generating correct and efficient code.

Interestingly, instruction tuning does not consistently yield improvements over the original CodeLLMs in either metric. Among the evaluated models, only Magicoder-S-DS-6.7B achieves a higher pass@1 compared to its original version, and only DeepseekCoder-6.7B-instruct and Magicoder-S-DS-6.7B show efficiency gains over their original versions (i.e., ECC > 0.5). We attribute this inconsistency to the inherent limitations of instruction tuning, which heavily depends on diverse and representative training datasets. In this study, EffiBench provides a single ground-truth solution per task, which may not be sufficient to effectively guide CodeLLMs in generating both efficient and correct code. Furthermore, instruction tuning typically employs next-token prediction as its training objective, which focuses on predicting subsequent tokens based on the context

Table 4. Correctness and Efficiency of Code Generated by CodeLLMs with Instruction tuning and CodeLLMs Fine-tuned by ACECode.

Model Name	Fine-Tune	ECC (%)	GET	Pass@1 (%)
CodeQwen1.5-7B-Chat	Instruction Tuning	47.1	3.03×10^{-4}	30.95
	ACECode	72.3	2.72×10^{-4} (↓10.23%)	46.77 (↑51.15%)
DeepseekCoder-6.7B-instruct	Instruction Tuning	56.7	1.21×10^{-3}	42.06
	ACECode	61.2	1.09×10^{-3} (↓9.92%)	48.41 (↑15.10%)
WaveCoder-Ultra-6.7B	Instruction Tuning	48.1	9.82×10^{-4}	41.26
	ACECode	65.4	8.66×10^{-4} (↓11.81%)	43.65 (↑5.79%)
Magicoder-S-DS-6.7B	Instruction Tuning	60.8	9.19×10^{-4}	50.00
	ACECode	69.7	7.06×10^{-4} (↓23.18%)	56.35 (↑12.70%)

As defined in Section 4.5, ECC measures the percentage of code generated by tuned CodeLLMs that has less execution time than that of original CodeLLMs. GET measures the geometric average execution time of the generated code. Pass@1 measures the percentage of generated code that passes the test cases at the first attempt.

rather than explicitly optimizing CodeLLMs for multi-objective and fine-grained goals. However, for the task of generating both correct and efficient code, the optimization goals encompass both code correctness and efficiency, necessitating an optimization process that extends beyond the capabilities of standard instruction tuning. Conversely, ACECode overcomes these challenges by utilizing a reinforcement learning-based fine-tuning method that 1) autonomously generates varied responses for each coding task to enable iterative training and 2) integrates multi-objective optimization directly into its reward system.

5.1.3 Comparison with PIE. We compare the performance of CodeLLMs fine-tuned by ACECode with those fine-tuned by PIE [34] in terms of code correctness and efficiency. The evaluation results are shown in Table 5.

CodeLLMs fine-tuned by ACECode consistently outperform PIE-tuned models in both code correctness and efficiency. In particular, ACECode enhances code correctness over PIE by a margin ranging from 2.92% to 14.41% in terms of pass@1. The runtime efficiency of code generated by ACECode-fine-tuned CodeLLMs surpasses that of PIE-tuned models by 5.31% to 11.45% in terms of GET score. The results suggest that ACECode is more effective in optimizing code efficiency and correctness than PIE.

Furthermore, we observe that though PIE improves the efficiency of generated code for all CodeLLMs (i.e., all ECC>0.5), it does not consistently enhance code correctness. Specifically, CodeQwen1.5-7B-Chat and WaveCoder-Ultra-6.7B show lower pass@1 scores than the vanilla models that did not undergo fine-tuning. This finding is aligned with existing studies [35] that highlight the trade-off between code efficiency and correctness in PIE. In contrast, ACECode simultaneously enhances both code efficiency and correctness, overcoming these limitations.

Table 5. Correctness and Efficiency of Code Generated by CodeLLMs fine-tuned with PIE and ACECode.

Model Name	Fine-Tune	ECC (%)	GET	Pass@1 (%)
CodeQwen1.5-7B-Chat	PIE	62.3	1.13×10^{-3}	40.88
	ACECode	72.3	1.07×10^{-3} (↓5.31%)	46.77 (↑14.41%)
DeepseekCoder-6.7B-instruct	PIE	56.1	9.17×10^{-4}	44.61
	ACECode	61.2	8.12×10^{-4} (↓11.45%)	48.41 (↑8.51%)
WaveCoder-Ultra-6.7B	PIE	59.7	1.65×10^{-3}	42.41
	ACECode	65.4	1.55×10^{-3} (↓6.06%)	43.65 (↑2.92%)
Magicoder-S-DS-6.7B	PIE	58.3	1.15×10^{-3}	52.46
	ACECode	67.9	1.07×10^{-3} (↓6.96%)	56.35 (↑7.42%)

Key Findings

Summary: The efficiency and correctness of code generated by CodeLLMs fine-tuned by ACECode are improved compared to the original CodeLLMs by 0.52% to 10.86% in terms of GET score and by 1.84% to 14.51% in terms of pass@1 for four state-of-the-art CodeLLMs. Furthermore, ACECode outperforms instruction tuning and PIE in terms of code correctness and efficiency, achieving improvements of up to 51.15% in pass@1 and 23.18% in GET score compared to instruction tuning, and up to 14.41% in pass@1 and 11.45% in GET score compared to PIE.

5.2 RQ2: Impact of ACECode Parameters on CodeLLMs Performance

We conduct ablation studies to investigate the impact of ACECode parameters on the performance of CodeLLMs fine-tuned by ACECode. We identify the following factors of ACECode that may affect its performance: (1) the training epoch number and (2) the number of CodeLLM responses.

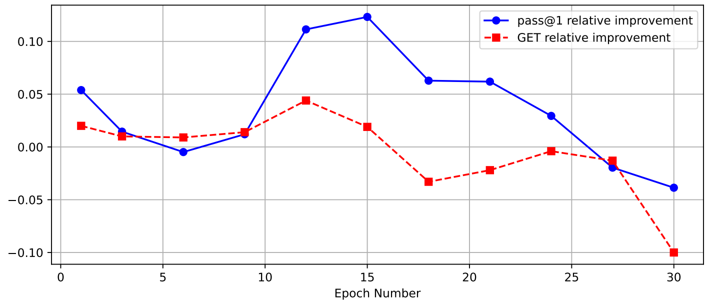

Fig. 3. Impact of Training Epoch Numbers on ACECode Performance. Pass@1 and GET relative improvements refer to the percentage of improvement in terms of Pass@1 and GET scores compared to the original CodeLLMs.

5.2.1 Number of Training Epoches. **Experiment Setting:** To investigate the impact of the training epoch on the performance of ACECode, we compare the performance of CodeQwen1.5-7B-Chat fine-tuned by ACECode with different training epoch numbers ranging from 1 to 30. We set the epoch interval to be 3.

Results: The evaluation results are shown in Figure 3. We observe that epoch number affects the performance of ACECode. Specifically, as the training epoch number increases from 1 to 18, the GET score of tuned CodeLLMs is consistently higher than that of the original CodeLLMs by 0.01 to 0.05. Similarly, the Pass@1 score of tuned CodeLLMs is also higher than the original CodeLLMs by up to 11.52% from 1 to 24 epoch except for the 6th epoch, which is 0.01% lower than the original CodeLLMs. The improvement variance may be due to the randomness of the RL training process. However, we observe that the performance of tuned CodeLLMs in terms of GET and Pass@1 starts to decrease and performs worse than the original CodeLLMs consistently after 18 and 24 epochs, respectively. We hypothesize that overfitting may occur when the number of training epochs is too large, which leads to a performance decrease of ACECode.

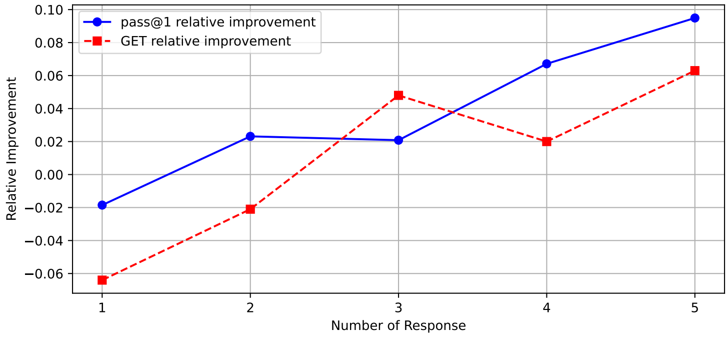

Fig. 4. Impact of Number of Responses on ACECode Performance. Pass@1 and GET relative improvements refer to the percentage of improvement in terms of Pass@1 and GET scores compared to the original CodeLLMs.

5.2.2 Number of CodeLLM Response. **Experiment Setting:** We investigate the impact of response number N on the performance of ACECode. A response number N indicates that ACECode generates N responses for each coding task in the training dataset, which leads the pairs of coding tasks and solutions that are used for RL training to be N times the size of the original dataset. We conduct experiments using N ranging from 1 to 5 for fine-tuning CodeQwen1.5-7B-Chat on ACECode.

Results: The evaluation results are shown in Figure 4. We can observe that both GET and Pass@1 scores of tuned CodeLLMs generally increase as the response number increases from 1 to 5, except for the GET score when the response number is 4 and the pass@1 score when the response number is 3. We attribute these slight declines to potential variability in RL training. Specifically, both GET and Pass@1 scores achieve the highest improvement when the response number is 5, indicating that the performance of ACECode is positively correlated with the response number and follows the trend of scaling law. However, the training time cost of ACECode is approximately linearly increasing with the response number, presenting a trade-off between the performance and training cost of ACECode.

6 Threats to Validity

The threats to external validity concern the generalizability of our findings. To mitigate this threat, we evaluated ACECode using four popular and state-of-the-art CodeLLMs on the Evalplus [24] benchmark. However, we acknowledge that our findings may not generalize to CodeLLMs of other sizes or architectures. In the future, we plan to extend our study to a broader range of CodeLLMs to further validate the effectiveness of ACECode.

The threats to internal validity of this study include the measurement of execution time and correctness of CodeLLM-generated code. To mitigate this threat, we run each code snippet with an adaptive repeated strategy to ensure the code snippets are executed multiple times and take the average execution time as the final result. Meanwhile, different from the previous benchmark EffiBench, which uses profiling tools to measure the execution time of code snippets, we follow the best practices in Python documentation for runtime benchmarking [3] and use *timeit* module. Furthermore, we take into account the library import overhead and exclude its time cost from the final result by leveraging the `timeit` module. On the other hand, to mitigate the threat of randomness in code correctness evaluation, we fix the temperature of LLM as 0 and top_p as 1 to force LLMs to generate identical code snippets for each coding task.

The threats to construct validity mainly lie in the selection of evaluation metrics. To mitigate this threat, we identify the deficiencies of existing evaluation metrics and propose two new evaluation metrics to better measure the performance of CodeLLMs.

7 Conclusion

In this paper, we propose ACECode, a reinforcement learning-based fine-tuning framework for CodeLLMs, to improve CodeLLMs' performance in terms of both the correctness and efficiency of their generated code. ACECode leverages execution feedback of the generated code to guide reinforcement learning-based fine-tuning of CodeLLMs, in which the reward is calculated based on the execution correctness and runtime efficiency of the generated code. We conduct experiments on four state-of-the-art CodeLLMs and evaluate the performance of CodeLLMs fine-tuned on ACECode in terms of code correctness and efficiency. The results show that CodeLLMs fine-tuned by ACECode outperform the original CodeLLMs in terms of code correctness and efficiency by from 1.84% to 14.51% in terms of pass@1 and from 0.52% to 10.86% in terms of GET score. In the future, we plan to extend our study to more CodeLLMs with different architectures, pre-training objectives, and parameter sizes.

References

[1] [n. d.]. Hugging Face. https://huggingface.co/. (Accessed on 10/25/2024).

[2] [n. d.]. Leetcode. https://leetcode.com/. (Accessed on 10/25/2024).

[3] [n. d.]. Python Documentation 3.9: Profile. https://docs.python.org/3.9/library/profile.html. (Accessed on 10/25/2024).

[4] [n. d.]. Python Documentation 3.9: timeit. https://docs.python.org/3.9/library/timeit.html. (Accessed on 10/25/2024).

[5] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[7] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

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

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

[10] Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2023. Safe rlhf: Safe reinforcement learning from human feedback. *arXiv preprint arXiv:2310.12773* (2023).

[11] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[12] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[13] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 71–80.

[14] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*. PMLR, 12098–12107.

[15] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).

[16] Jian Hu, Xibin Wu, Weixun Wang, Xianyu, Dehao Zhang, and Yu Cao. 2024. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. *arXiv preprint arXiv:2405.11143* (2024).

[17] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Jie M Zhang, Heming Cui, and Zhijiang Guo. 2024. SOAP: Enhancing Efficiency of Generated Code via Self-Optimization. *arXiv preprint arXiv:2405.15189* (2024).

[18] Dong Huang, Jie M Zhang, Yuhao Qing, and Heming Cui. 2024. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. *arXiv preprint arXiv:2402.02037* (2024).

[19] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[20] Jack Kelly. 2024. AI Writes Over 25% Of Code At Google—What Does The Future Look Like For Software Engineers? — forbes.com. https://www.forbes.com/sites/jackkelly/2024/11/01/ai-code-and-the-future-of-software-engineers/. [Accessed 05-11-2024].

[21] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.

[22] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2404.00599* (2024).

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

[24] 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*. https://openreview.net/forum?id=1qvx610Cu7

[25] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).

[26] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[27] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. 103–107.

[28] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[29] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

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

[32] Tianhao Shen, Renren Jin, Yufei Huang, Chuang Liu, Weilong Dong, Zishan Guo, Xinwei Wu, Yan Liu, and Deyi Xiong. 2023. Large language model alignment: A survey. *arXiv preprint arXiv:2309.15025* (2023).

[33] Jieke Shi, Zhou Yang, and David Lo. 2024. Efficient and Green Large Language Models for Software Engineering: Vision and the Road Ahead. *arXiv preprint arXiv:2404.04566* (2024).

[34] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867* (2023).

[35] Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. 2024. ECCO: Can We Improve Model-Generated Code Efficiency Without Sacrificing Functional Correctness? *arXiv preprint arXiv:2407.14044* (2024).

[36] Bo Wang, Sirui Lu, Yingfei Xiong, and Feng Liu. 2021. Faster mutation analysis with fewer processes and smaller overheads. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 381–393.

[37] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[38] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[39] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Aligning large language models with human: A survey. *arXiv preprint arXiv:2307.12966* (2023).

[40] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).

[41] Shusheng Xu, Wei Fu, Jiaxuan Gao, Wenjie Ye, Weilin Liu, Zhiyu Mei, Guangju Wang, Chao Yu, and Yi Wu. 2024. Is dpo superior to ppo for llm alignment? a comprehensive study. *arXiv preprint arXiv:2404.10719* (2024).

[42] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187* (2023).

[43] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).