EFFI-CODE: Enhancing Code Generation in Large Language Models through Efficiency-Aware Fine-tuning

Anonymous Author(s)
Affiliation
Address
email

Abstract

As large language models (LLMs) play an increasingly important role in code generation, enhancing both correctness and efficiency has become crucial. Current methods primarily focus on correctness, often overlooking efficiency. To address this gap, we introduce EFFI-CODE to improve both aspects by fine-tuning LLMs on a high-quality dataset comprising correct and efficient code samples. Our methodology involves leveraging multiple LLMs to generate diverse candidate code solutions for various tasks across different programming languages. We then evaluate these solutions by directly measuring their execution time and memory usage through local execution. The code solution with the lowest execution time and memory consumption is selected as the final output for each task. Experimental results demonstrate significant improvements when fine-tuning with EFFI-CODE. For instance, Qwen2.5-Coder-7B-Instruct's pass@1 score increases from 44.8% to 57.7%, while the average execution time for correct tasks decreases by 48.4%. EFFI-CODE offers a scalable and effective solution for advancing AI-driven code generation, benefiting both software development and computational problem-solving.

1 Introduction

Large language models (LLMs) have recently made significant strides across various tasks [OpenAI, 2023, Anil et al., 2023, Anthropic, 2024, Meta, 2024], including code-related applications like code completion [Chen et al., 2021, Austin et al., 2021], debugging [Haque et al., 2022, Chen et al., 2023], and translation[Rozière et al., 2020, Ahmad et al., 2023]. Before deploying LLMs into integrated development environments (IDEs) as tools, it is crucial to ensure that the generated code meets the required efficacy standards. To address this, researchers have explored various datasets to fine-tune LLMs, thereby improving the efficacy of LLM-generated code [Ouyang et al., 2022, Wei et al., 2022]. For example, Code Alpaca [Chaudhary, 2023] utilized the Self-Instruct framework [Wang et al., 2023] to synthesize data, while WizardCoder [Luo et al., 2024] employed the Evol-Instruct technique [Xu et al., 2024] to generate heuristic prompts for diverse solutions. Additionally, OSS-Instruct [Wei et al., 2024b] created new coding problems using open-source snippets with LLMs, and Octopack [Muennighoff et al., 2024] focused on curating high-quality Git commit messages that resemble natural language instructions. These fine-tuning efforts have led to increased correctness in LLM-generated code.

However, existing works primarily focus on enhancing the correctness of LLM-generated code while neglecting to optimize its efficiency. As a result, the efficiency of such code often falls short compared to canonical solutions written by human developers. Recent studies [Shi et al., 2024, Niu et al., 2024,

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

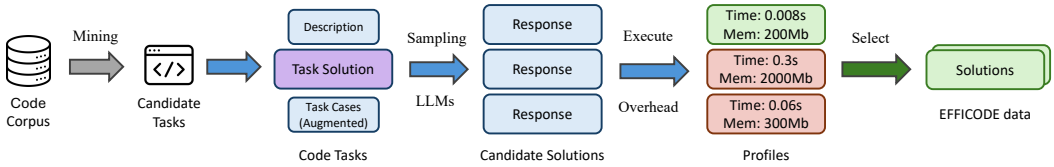

Figure 1: Overview of the construction pipeline for EFFI-CODE: We begin by collecting the initial EFFI-CODE from different open-source datasets. Starting with the original code, we require multiple LLMs to generate candidate solutions, using test cases to profile execution overhead, and use the most efficient solution generated by LLMs as the solution for each task. We then have our final fine-tuning dataset, EFFI-CODE, which consists of optimized code and rich metadata, designed to train models for generating efficient code.

Du et al., 2024, Huang et al., 2024a] also point out that LLM-generated code typically exhibits lower efficiency in execution time and memory usage. For instance, on the EffiBench benchmark [Huang et al., 2024b], even the most advanced LLMs, such as GPT-4-Turbo, produced less efficient code, with average and worst-case execution times being 1.69 and 45.49 times longer than those of canonical solutions, respectively. Efficiency is crucial because inefficient code consumes more computational resources, leading to higher energy consumption and increased operational costs. This is particularly important in the context of sustainability, as the demand for computing power continues to grow, and reducing the environmental impact of large-scale computations becomes a pressing concern. Furthermore, inefficient code may be impractical for use in resource-constrained environments, such as mobile devices or embedded systems, where both energy and processing power are limited. This underscores the urgent need to develop new methods that can enhance both the **correctness** and **efficiency** of LLM-generated code.

In this paper, we introduce the dataset EFFI-CODE, aimed at fine-tuning LLMs to improve both code efficiency and correctness. We begin by aggregating source code from existing open-source datasets. This is followed by a rigorous preprocessing and cleaning process, coupled with the generation of test cases for each task to evaluate code efficiency. We leverage multiple LLMs to generate diverse candidate code solutions for various tasks across different programming languages. We then evaluate these solutions by directly measuring their execution time and memory usage through local execution. Code solutions with the lowest execution time and memory consumption are selected as the final output. The resulting optimized code, along with its associated metadata, forms EFFI-CODE, which serves as a high-quality resource for training LLMs.

Extensive experiments demonstrate that fine-tuning LLMs with EFFI-CODE improves both correctness and efficiency. For example, the fine-tuned Qwen2.5-Coder-7B-Instruct [DeepSeekAI, 2023] increases the pass@1 from 44.8% and 76.2% to 57.7% and 78.0% on EffiBench and HumanEvalPlus, while also reducing the average execution time from 0.31 seconds to 0.16 seconds — representing a 48.4% reduction in execution time overhead on EffiBench. Compared to PIE [Shypula et al., 2024], which increases the pass@1 from 12.2% to 19.5% on HumanEvalPlus, the pass@1 of CodeLlama-7B [Rozière et al., 2023] fine-tuned with EFFI-CODE further increases to 31.1%. In addition, EFFI-CODE decreases the execution time by 46.2% while PIE decreases it by 23.1%. We will fully open-source EFFI-CODE and the source code to facilitate research. To conclude, this paper makes the contributions:

- We propose a versatile framework for constructing code generation datasets with efficient solutions, adaptable to various programming languages and sources.
- We introduce EFFI-CODE, to the best of our knowledge, it is the first instruction-tuning dataset designed to improve the efficiency of LLM-generated code, facilitating fine-tuning for more efficient code generation.
- We fine-tune various widely used LLMs using EFFI-CODE, demonstrating improvements in both correctness and efficiency.

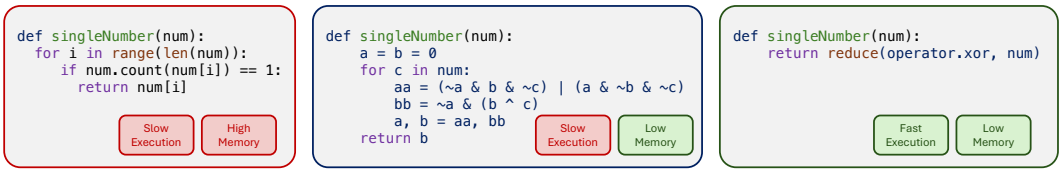

Figure 2: Examples of codes with levels of efficiency: The first solution features high memory usage and long execution time, the second achieves lower memory usage but still has a long execution time, and the third is optimized for both low memory usage and short execution time.

2 Related Works

2.1 LLMs for Code

The increasing popularity of LLMs for code generation has coincided with the growing availability of open-source code repositories and the need to boost developer productivity. Initial efforts focused on training models specifically for coding tasks, such as CodeT5 [Wang et al., 2021], AlphaCode [Li et al., 2022], CodeGen [Nijkamp et al., 2023], InCoder [Fried et al., 2023], StarCoder [Li et al., 2023a], SantaCoder [Allal et al., 2023], and DeepSeek-Coder [DeepSeekAI, 2023]. Contrastingly, models such as Codex [Chen et al., 2021] and CodeLlama [Rozière et al., 2023] represent a subsequent stride, being fine-tuned from foundation models [Brown et al., 2020, Touvron et al., 2023]. These code LLMs have been applied to various tasks, including code generation [Chen et al., 2021, Dai et al., 2024], program repair [Haque et al., 2022, Jiang et al., 2023], automated testing [Lemieux et al., 2023, Deng et al., 2023], code translation [Rozière et al., 2020, Ahmad et al., 2023], type prediction [Mir et al., 2022, Wei et al., 2023], and code summarization [Hasan et al., 2021, Ahmed and Devanbu, 2022]. While LLMs have achieved impressive results in code generation tasks like HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021], their efficiency has received less attention. Recent studies [Shi et al., 2024, Huang et al., 2024b, Niu et al., 2024] have shown that LLM-generated code exhibits lower efficiency in terms of execution time and memory usage compared to canonical solutions. These findings highlight the need for further research and development to improve the efficiency of LLM-generated code. In this work, we propose the first fine-tuning method that significantly improves both the efficiency and correctness of code generated by various LLMs.

2.2 Instruction Tuning for Code

Instruction tuning has proven effective in enhancing the usability and overall performance of LLMs across various language tasks [Ouyang et al., 2022, Wei et al., 2022, Zhao et al., 2024]. This approach has been extended to the domain of code generation. The core challenge is the acquisition of high-quality instructional data, which is often labor-intensive. To address this, recent research has focused on developing methods to generate synthetic instruction data. Studies have shown that textbook-quality synthetic data alone can improve a model's coding and reasoning capabilities [Gunasekar et al., 2023, Li et al., 2023b]. One early effort was Self-Instruct [Wang et al., 2023], which utilized LLMs to generate synthetic instruction-response pairs using carefully crafted prompts. The same LLM was then instruction-tuned on this synthetic data. Code Alpaca [Chaudhary, 2023] applied the Self-Instruct approach with GPT models, tailoring it specifically for code generation, editing, and optimization tasks. Building upon this, WizardCoder [Luo et al., 2024] adapted the Evol-Instruct technique [Xu et al., 2024] to the coding domain by designing heuristic prompts to create more complex and diverse synthetic data. OSS-Instruct [Wei et al., 2024b] took a different approach by leveraging LLMs to automatically generate new coding problems inspired by random code snippets from open-source repositories. In contrast, Octopack [Muennighoff et al., 2024] focused on collecting and filtering high-quality Git commit messages that resemble natural language instructions. While these existing methods primarily emphasize generating correct code, EFFI-CODE explores the use of fine-tuning to improve code efficiency. Our method is orthogonal to existing synthetic techniques, offering the potential for combination to further enhance the coding capabilities of LLMs.

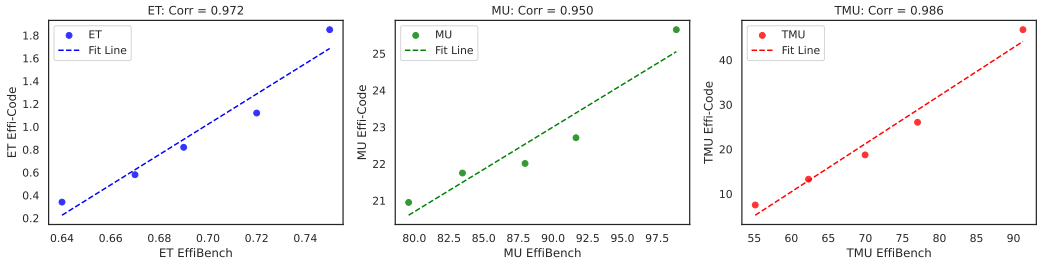

Figure 3: Correlation of the efficiency of the automatically generated code and the LLM code train set.

3 EFFI-CODE: Fine-Tuning For Efficiency

3.1 Preliminary Study

We begin by investigating how the efficiency of training data influences the efficiency of code generated by LLMs. Following prior works [Huang et al., 2024b], we evaluate code efficiency using three metrics: Execution Time (ET), Max Memory Usage (MU), and Total Memory Usage (TMU). Our hypothesis is that training LLMs on efficient code will lead to the generation of more efficient code. To test this hypothesis, we synthesized multiple training datasets with varying levels of efficiency and used them to train different LLMs. The efficiency of the generated code was then measured in a controlled environment using ET, MU, and TMU. The training datasets included both efficient and inefficient code samples to ensure a comprehensive range of efficiencies. The results, presented in Figure 3, reveal strong positive correlations between the efficiency of the training data and the efficiency of the generated code. Specifically, the correlation for ET is 0.972, for MU it is 0.950, and for TMU it is 0.986. These high correlation coefficients indicate that as the efficiency of the training data increases, so does the efficiency of the generated code. This study demonstrates that training LLMs on efficient code significantly enhances the efficiency of the generated code. The strong correlations across all three metrics support the hypothesis that the efficiency of the training data is a critical factor in improving the performance of LLM-generated code. These findings inspire further exploration of specific techniques for optimizing training datasets to maximize code efficiency.

3.2 Dataset Construction

Curation Process Figure 1 illustrates an overview of the process for constructing the EFFI-CODE dataset for fine-tuning. The first step involves collecting candidate code generation tasks from nine open-source datasets available on the HuggingFace platform[1]. For each task, we aim to construct a more efficient solution compared to the initial solutions provided by the open-source datasets. Our approach shares similarities with existing works Du et al. [2024], Huang et al. [2024a], where researchers execute LLM-generated code locally and analyze the execution time and memory usage. However, our construction pipeline differs in that it utilizes multiple LLMs (e.g., DeepSeek-Coder and GPT-4o) to generate multiple candidate code solutions for each task in our candidate task set. We then directly calculate the execution time and memory usage for each generated code solution by executing them in local environments. The code with the lowest execution time and memory usage is selected as the final code for each task. For example, as shown in Figure 2, we directly select the code on the right as the final code.

Data Sources We collect the candidate tasks from the open-source code LLM training sets, which include SelfCodeAlign (SelfCodeAlign; Wei et al. 2024a), CodeFeedback-Filtered-Instruction (Code-Feed; MAP 2023), Tested-143k-Python-Alpaca (Alpaca; Vezora 2023), Glaive-Code-Assistant (Glaive; Computer 2023), Magicoder-Evol-Instruct-110K (Evol-Ins; UIUC 2023a), Dolphin-Coder (Dolphin; Computations 2023), Magicoder-OSS-Instruct-75K (Oss-Ins; UIUC 2023b), Self-OSS-Instruct-SC2-Exec-Filter-50K (Self-Oss; BigCode 2023), and Apps [Hendrycks et al., 2021]. To collect the candidate tasks, all Python, C++, Java, Rust, and Go functions are extracted from the

[1] `https://huggingface.co/docs/datasets`

Table 1: Distribution of tasks in the constructed EFFI-CODE for different programming languages.

Dataset	APPS	Alpaca	CodeFeed	Glaive	Evol-Ins	Dolphin	Oss-Ins	Self-Oss	SelfCodeAlign	Total
Python	1001	2920	1387	32	1250	1958	76	827	24038	33489
CPP	-	3	1257	2675	3439	1186	2985	2	-	11547
Java	-	1	2082	3278	4692	1746	2927	-	-	14726
Rust	-	-	26	187	467	500	3090	-	-	4270
Go	-	1	47	277	776	549	28	-	-	1678

aforementioned open-source datasets. Following the filtering instructions of SelfCodeAlign Wei et al. [2024a], a series of filtering rules are applied to ensure the code quality of the candidate tasks. After applying the filtering process, a total of 65k tasks were collected from an initial pool of about 790k candidate tasks.

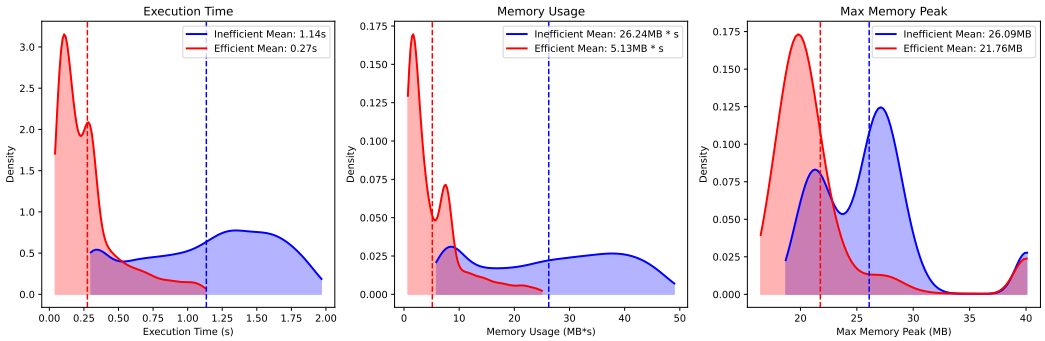

Figure 4: Efficiency distribution of the Python subset collected from Hugging Face. The figure shows the distribution of execution time, memory usage, and max memory peak for both inefficient (task-provided solution) and efficient solutions in the EFFI-CODE. The inefficient solutions have higher overheads for all three metrics than the efficient ones.

3.3 Dataset Statistics

As shown in Table 1, coding problems in EFFI-CODE have been collected from nine datasets, resulting in a total of 65,710 tasks across five programming languages: Python, C++, Java, Rust, and Go. The dataset encompasses a diverse range of coding challenges, ensuring a comprehensive coverage of various programming concepts and problem-solving techniques. Python has the highest representation in EFFI-CODE, with 33,489 tasks sourced from all nine datasets. This extensive collection of Python tasks allows for effective fine-tuning of LLMs to generate efficient and optimized Python code. C++ and Java also have significant contributions, with 11,547 and 14,726 tasks, respectively. These tasks are primarily sourced from CodeFeed, Glaive, Evol-Ins, Dolphin, and Oss-Ins datasets, providing a robust foundation for fine-tuning LLMs in these popular programming languages. Rust and Go, although having relatively fewer tasks compared to Python, C++, and Java, still have a substantial presence in EFFI-CODE. With 4,270 Rust tasks and 1,678 Go tasks, the dataset enables fine-tuning of LLMs to generate efficient code in these modern and rapidly growing programming languages.

Figure 4 illustrates the efficiency distribution of the dataset for three key metrics: execution time, memory usage, and max memory peak, which compares the distribution of these metrics for both inefficient (canonical solutions provided by the nine datasets) and efficient solutions in the EFFI-CODE. For execution time, the inefficient solutions have a mean value of 1.14s, while the efficient solutions have a significantly lower mean of 0.31s, which indicates that the optimization process has successfully reduced the execution time of the code, resulting in more efficient solutions. Similarly, the memory usage and max memory peak also show a notable difference between inefficient and efficient solutions. For example, inefficient solutions have a mean memory usage of 26.50MBs, while efficient solutions have a much lower mean of 6.03MBs.

The efficiency distribution visualization highlights the effectiveness of the optimization process in creating more efficient solutions across all three metrics. By carefully curating tasks through

the multi-step cleaning process and applying SOAP optimization, we have created a dataset that is valuable for training models to generate efficient code. EFFI-CODE provides a diverse range of optimized coding problems, enabling researchers and practitioners to advance the field of code optimization using LLMs.

4 Experiment

Datasets and Models We evaluate the efficiency and correctness of LLM-generated code on five code generation benchmarks, i.e., EffiBench Huang et al. [2024b], EvalPlus (HumanEvalPlus and MBPPPlus) Liu et al. [2024], DS-1000 Lai et al. [2023], EvoEval Xia et al. [2024], and HumanEval-X Zheng et al. [2023]. We finetune eight open-source LLMs with EFFI-CODE, including CodeLlama-7b-bf, DeepSeek-Coder-6.7B base and instruct model [DeepSeekAI, 2023], Qwen2.5-Code-7B base and instruct model [Hui et al., 2024], and Qwen2.5-Coder (1.5B, 3B, and 14B).

Fine-tuning Setup We use Llama-factory [Zheng et al., 2024] to fully fine-tune all LLMs with the same setup and train the models using EFFI-CODE. The maximum sequence length is set to 2048 tokens. We use a batch size of 128 and set the learning rate to 5e-6 with a cosine learning rate scheduler and a warmup ratio of 0.03. We fine-tune all LLMs for 4 epochs under the bf16 data type.

Table 2: Code efficiency and pass@1 of LLMs trained with EFFI-CODE on EffiBench using greedy decoding. The percentage in the brackets indicates the extent of the reduction for each respective item. `Overlap` means the percentage of correct tasks addressed by both EFFI-CODE finetuned LLM and original LLM in total tasks of the dataset. We provide a case example in Figure 5 to demonstrate how EFFI-CODE fine-tuned LLM improves the efficiency of LLM-generated code.

Model	ET (s) ↓	NET ↓	MU (Mb) ↓	NMU ↓	TMU (Mb*s) ↓	NTMU ↓	Overlap (%) ↑	Pass@1 (%) ↑
CodeLlama-7b-hf	0.24	1.48	141.91	4.78	149.23	73.82	8.2	15.0
+ SFT	0.20 (16.7%)	1.27 (14.2%)	80.86 (43.0%)	2.26 (52.7%)	87.09 (41.6%)	42.66 (42.2%)	8.2	17.6
deepseek-coder-6.7b-base	0.37	2.62	36.94	1.04	17.26	2.74	47.1	54.4
+ SFT	0.21 (43.2%)	1.42 (45.8%)	37.11 (-0.5%)	1.04 (0.0%)	11.97 (30.6%)	2.10 (23.4%)	47.1	59.3
deepseek-coder-6.7b-instruct	0.34	2.56	47.26	1.45	30.05	9.97	36.0	44.4
+ SFT	0.22 (35.3%)	1.71 (33.2%)	36.31 (23.2%)	1.00 (31.0%)	9.48 (68.5%)	2.11 (78.8%)	36.0	51.7
Qwen2.5-Coder-7B-Instruct	0.31	2.35	31.66	1.00	11.00	2.15	37.2	44.8
+ SFT	0.16 (48.4%)	1.12 (52.3%)	31.67 (-0.0%)	1.00 (0.0%)	8.28 (24.7%)	1.18 (45.1%)	37.2	57.7
Qwen2.5-Coder-1.5B	0.40	2.95	35.34	1.03	15.68	3.51	27.1	39.6
+ SFT	0.22 (45.0%)	1.60 (45.8%)	34.58 (2.2%)	1.00 (2.9%)	9.09 (42.0%)	1.98 (43.6%)	27.1	41.7
Qwen2.5-Coder-3B	0.43	2.73	48.73	1.00	33.88	2.59	16.9	31.2
+ SFT	0.23 (46.5%)	1.60 (41.4%)	49.18 (-0.9%)	1.00 (0.0%)	18.31 (46.0%)	1.98 (23.6%)	16.9	34.2
Qwen2.5-Coder-7B	0.26	1.81	38.88	1.01	18.63	3.01	41.4	50.1
+ SFT	0.17 (34.6%)	1.23 (32.0%)	38.61 (0.7%)	1.00 (1.0%)	10.82 (41.9%)	1.32 (56.1%)	41.4	57.3
Qwen2.5-Coder-14B	0.36	2.73	32.41	1.00	12.59	2.57	50.1	57.5
+ SFT	0.15 (58.3%)	1.14 (58.2%)	32.41 (0.0%)	1.00 (0.0%)	6.80 (46.0%)	1.23 (52.1%)	50.1	63.6

Table 3: Code efficiency and pass@1 of LLMs trained with EFFI-CODE on HumanEval-EvalPlus.

Model	ET (s) ↓	NET ↓	MU (Mb) ↓	NMU ↓	TMU (Mb*s) ↓	NTMU ↓	Overlap (%) ↑	Pass@1 (%) ↑
HumanEvalPlus								
deepseek-coder-6.7b-base	0.43	1.04	67.45	1.00	28.23	1.02	7.3	7.3
+ SFT	0.42 (2.3%)	1.00 (3.8%)	67.37 (0.1%)	1.00 (0.0%)	27.90 (1.2%)	1.02 (0.0%)	7.3	64.6
deepseek-coder-6.7b-instruct	0.54	2.27	61.64	0.98	20.18	2.30	42.1	47.6
+ SFT	0.37 (31.5%)	1.45 (36.1%)	61.58 (0.1%)	0.98 (0.0%)	16.48 (18.3%)	1.78 (22.6%)	42.1	72.6
Qwen2.5-Coder-7B	0.35	1.23	61.77	0.98	15.25	1.39	36.6	40.2
+ SFT	0.29 (17.1%)	0.96 (22.0%)	61.70 (0.1%)	0.98 (0.0%)	12.18 (20.1%)	0.96 (30.9%)	36.6	78.7
Qwen2.5-Coder-7B-Instruct	0.52	2.05	63.38	0.99	20.17	1.96	67.7	76.2
+ SFT	0.32 (38.5%)	1.08 (47.3%)	63.35 (0.0%)	0.99 (0.0%)	15.15 (24.9%)	1.16 (40.8%)	67.7	78.0
MBPPPlus								
deepseek-coder-6.7b-base	0.49	1.64	58.90	1.00	17.27	1.62	55.3	63.2
+ SFT	0.31 (36.7%)	0.97 (40.9%)	58.99 (-0.2%)	1.00 (0.0%)	10.27 (40.5%)	0.96 (40.7%)	55.3	65.9
deepseek-coder-6.7b-instruct	0.43	1.65	59.03	1.00	14.39	1.65	59.0	65.3
+ SFT	0.31 (27.9%)	1.01 (38.8%)	58.97 (0.1%)	1.00 (0.0%)	10.35 (28.1%)	1.02 (38.2%)	59.0	67.5
Qwen2.5-Coder-7B	0.48	1.70	58.89	0.99	17.00	1.79	59.5	60.1
+ SFT	0.31 (35.4%)	0.96 (43.5%)	58.98 (-0.2%)	0.99 (0.0%)	10.33 (39.2%)	0.94 (47.5%)	59.5	63.2
Qwen2.5-Coder-7B-Instruct	0.46	1.68	64.90	1.00	23.99	1.66	63.2	68.0
+ SFT	0.30 (34.8%)	0.96 (42.9%)	68.31 (-5.3%)	1.00 (0.0%)	16.91 (32.4%)	0.95 (42.8%)	63.2	70.6

4.1 Evaluation of Python Code

To comprehensively demonstrate the efficiency and correctness of automatically generated code by LLMs with the SFT of EFFI-CODE, we first provide the evaluation of the LLMs in generating Python code, where LLMs are asked to create Python code based on natural language or function signatures with docstrings.

EffiBench is a benchmark used to measure the efficiency and correctness of LLM-generated code in LeetCode tasks. To ensure that the efficiency of LLM-generated code can be measured, the authors construct 100 tests for each task to provide enough testing time. As shown in Table 2, we can observe that for all LLMs, the efficiency of the LLM-generated code has been improved after fine-tuning with EFFI-CODE. For example, the average execution time (ET) for the correct code generated by Qwen2.5-Coder-7B-Instruct and its EFFI-CODE fine-tuned version decreases from 0.31 (s) to 0.16 (s), a reduction of 48.4%. Similarly, the total memory usage (TMU) of LLM-generated code also shows significant decreases. For instance, the TMU of DeepSeek-Coder-6.7B-Instruct decreases from 30.05 (Mb*s) to 9.48 (Mb*s), a larger reduction than the decrease in average execution time for the correct code generated by the same model, which only reduces by 35.3% from 0.34 (s) to 0.22 (s). This indicates that during code generation, both the execution time and memory usage of the EFFI-CODE fine-tuned-LLM-generated code have been improved compared to the code generated by the original models. Furthermore, the memory usage/peak (MU) of DeepSeek-Coder-6.7B-Instruct generated code decreases from 47.26 (Mb) to 36.31 (Mb), a reduction of 23.2%, ensuring that the LLM-generated code can be deployed in memory-constrained scenarios such as embedded systems or edge devices.

Interestingly, we observe that compared to the MU, ET is more widely optimized across all models. This suggests that EFFI-CODE fine-tuning has a more significant impact on reducing the execution time of the generated code than on reducing its memory footprint. Nevertheless, the improvements in execution time and memory usage demonstrate the effectiveness of EFFI-CODE in enhancing the efficiency of LLM-generated code.

Table 4: Code efficiency and pass@1 of Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct fine-tuned using SFT with the EFFI-CODE for the EvoEval dataset.

Model	ET (s) ↓	NET ↓	MU (Mb) ↓	NMU ↓	TMU (Mb*s) ↓	NTMU ↓	Overlap (%) ↑	Pass@1 (%) ↑
EvoEval_tool_use								
Qwen2.5-Coder-7B-Instruct	0.35	1.80	59.11	1.00	11.22	1.74	33.0	43.0
+ SFT	0.20 (42.9%)	0.98 (45.6%)	59.08 (0.1%)	1.00 (0.0%)	6.55 (41.6%)	0.99 (43.1%)	33.0	53.0
deepseek-coder-6.7b-instruct	0.41	1.83	74.88	1.00	31.51	1.74	44.0	54.0
+ SFT	0.24 (41.5%)	0.99 (45.9%)	74.77 (0.1%)	1.00 (0.0%)	26.50 (15.9%)	1.00 (42.5%)	44.0	55.0
EvoEval_subtle								
Qwen2.5-Coder-7B-Instruct	0.40	1.49	70.46	1.00	29.15	1.46	48.0	55.0
+ SFT	0.33 (17.5%)	1.11 (25.5%)	70.47 (-0.0%)	1.00 (0.0%)	28.30 (2.9%)	1.17 (19.9%)	48.0	72.0
deepseek-coder-6.7b-instruct	0.45	1.97	59.06	0.99	17.01	2.00	50.0	56.0
+ SFT	0.30 (33.3%)	1.32 (33.0%)	58.93 (0.2%)	0.99 (0.0%)	11.19 (34.2%)	1.31 (34.5%)	50.0	69.0
EvoEval_creative								
Qwen2.5-Coder-7B-Instruct	0.51	2.16	62.71	1.00	24.21	2.39	32.0	44.0
+ SFT	0.37 (27.5%)	1.45 (32.9%)	62.69 (0.0%)	1.00 (0.0%)	21.10 (12.8%)	1.82 (23.8%)	32.0	44.0
deepseek-coder-6.7b-instruct	0.42	1.71	61.88	1.00	15.91	1.62	27.0	31.0
+ SFT	0.26 (38.1%)	0.99 (42.1%)	61.75 (0.2%)	1.00 (0.0%)	10.86 (31.7%)	0.98 (39.5%)	27.0	41.0

HumanEvalPlus and MBPPPlus As shown in Table 3, we observe that almost all LLMs achieve better efficiency and higher correctness after being fine-tuned with EFFI-CODE. Take HumanEvalPlus as an example, the average execution time (ET) for correct code generated by Qwen2.5-Coder-7B-Instruct and its fine-tuned version decreases from 0.52 (s) to 0.32 (s), a reduction of 38.5%. These improvements demonstrate the effectiveness of EFFI-CODE in optimizing the efficiency of LLM-generated code. Moreover, the pass@1 of Qwen2.5-Coder-7B increases from 40.2% to 78.7%, an improvement of 38.5% after fine-tuning with EFFI-CODE. This indicates that the fine-tuned models not only generate more efficient code but also produce correct code more frequently. Similar to the results in HumanEvalPlus, the efficiency and correctness of EFFI-CODE fine-tuned LLMs also improve in the MBPPPlus dataset. For instance, the average execution time for correct code generated by deepseek-coder-6.7b-base decreases by 36.7%, and the total memory usage (TMU) decreases by 40.5% after fine-tuning.

Table 5: Code efficiency and pass@1 of Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct fine-tuned using SFT with the EFFI-CODE for the DS-1000 dataset.

Model	ET (s) ↓	NET ↓	MU (Mb) ↓	NMU ↓	TMU (Mb*s) ↓	NTMU ↓	Overlap (%) ↑	Pass@1 (%) ↑
Qwen2.5-Coder-7B-Instruct	1.2668	1.0102	447.7101	1.0148	6.831366	1.0175	9.30	17.00
+ SFT	1.2593 (0.8%)	1.0041 (1.0%)	447.0711 (0.1%)	1.0134 (0.0%)	6.7694 (0.9%)	1.0081 (1.0%)	9.30	35.70
deepseek-coder-6.7b-instruct	1.3154	1.0536	450.2792	1.0206	7.204722	1.0731	29.10	37.50
+ SFT	1.2587 (4.5%)	1.0082 (3.8%)	441.7553 (1.9%)	1.0013 (2.0%)	6.8022 (5.6%)	1.0132 (5.6%)	29.10	37.50

Table 6: Code efficiency and pass@1 of Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct fine-tuned using SFT with the EFFI-CODE for the HumanEval-X (CPP and Java) dataset.

Model	ET (s) ↓	NET ↓	MU (Mb) ↓	NMU ↓	TMU (Mb*s) ↓	NTMU ↓	Overlap (%) ↑	Pass@1 (%) ↑
CPP								
Qwen2.5-Coder-7B-Instruct	0.0015	1.3973	1.5223	1.9489	0.000013	5.195024	4.27	7.32
+ SFT	0.0009 (37.1%)	1.0138 (27.4%)	0.8739 (42.6%)	1.0585 (45.7%)	0.000006 (53.8%)	2.784936 (46.4%)	4.27	48.17
deepseek-coder-6.7b-instruct	0.0015	1.2417	1.2991	1.0731	0.000010	2.078790	4.27	14.63
+ SFT	0.0008 (45.2%)	0.5312 (57.2%)	0.6496 (50.0%)	0.4289 (60.0%)	0.000006 (40.0%)	0.543047 (73.9%)	4.27	40.24
Java								
Qwen2.5-Coder-7B-Instruct	-	-	-	-	-	-	-	-
+ SFT	0.0082	0.2037	8.5926	0.1918	0.002996	0.1859	-	57.93
deepseek-coder-6.7b-instruct	0.0076	0.1921	8.1062	0.1789	0.002731	0.180034	6.71	14.63
+ SFT	0.0048 (36.5%)	0.1231 (35.9%)	4.0142 (50.5%)	0.0908 (49.2%)	0.001900 (30.4%)	0.118972 (33.9%)	6.71	57.93

Table 7: Efficiency comparison of different methods on the HumanEvalPlus dataset. We use the fine-tuned CodeLlama-7b-hf by PIE and Mercury as the baselines to measure the improvement of EFFI-CODE fine-tuned version.

Method	ET	NET	MU	NMU	TMU	NTMU	overlapped	pass@1
PIE	0.30 (23.1%)	1.47 (24.2%)	61.39 (0.5%)	1.00 (0.0%)	11.28 (11.7%)	1.68 (8.2%)	9.8	19.5
Ours	0.21 (46.2%)	1.03 (46.9%)	61.33 (0.6%)	1.00 (0.0%)	7.17 (43.9%)	1.04 (43.2%)	9.8	31.1
Mercury	0.31 (20.5%)	1.51 (22.2%)	61.94 (-0.4%)	1.00 (0.0%)	10.24 (19.9%)	1.47 (19.7%)	4.3	9.1
Ours	0.21 (46.2%)	1.01 (47.9%)	61.73 (-0.1%)	1.00 (0.0%)	6.95 (45.6%)	0.98 (46.4%)	4.3	31.1

EvoEval includes 828 programming problems created by prompting GPT-4 to evolve original HumanEval tasks across 5 semantic-altering and 2 semantic-preserving benchmarks, each of which has 100 problems. We conduct experiments on the Tool_Use, Subtle, and Creative benchmarks to evaluate the performance of Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct fine-tuned with EFFI-CODE. As shown in Table 4, both models demonstrate significant improvements in code efficiency after fine-tuning with EFFI-CODE. For the Tool_Use benchmark, the average execution time (ET) for correct code generated by Qwen2.5-Coder-7B-Instruct decreases by 42.9%, and the total memory usage (TMU) decreases by 41.6% after fine-tuning. Similarly, deepseek-coder-6.7b-instruct achieves a 41.5% reduction in ET and a 15.9% reduction in TMU. In the Subtle benchmark, after fine-tuning, Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct exhibit 17.5% and 33.3% reductions in ET, respectively, after fine-tuning. The normalized total memory usage (NTMU) also decreases by 19.9% and 34.5% for the two models. Moreover, the pass@1 rate improves significantly, with Qwen2.5-Coder-7B-Instruct increasing from 55.0% to 72.0% and deepseek-coder-6.7b-instruct increasing from 56.0% to 69.0%. For the Creative benchmark, Qwen2.5-coder-7B-instruct and deepseek-coder-6.7b-instruct achieve 27.5% and 38.1% reductions in ET, respectively, after fine-tuning. The NTMU also decreases by 23.8% and 39.5% for the two models. The pass@1 rate remains the same for Qwen2.5-Coder-7B-Instruct at 44.0% but improves from 31.0% to 41.0% for deepseek-coder-6.7b-instruct.

4.2 Data Science Programming

DS-1000 is a data science benchmark consisting of 1000 realistic challenges across 7 popular Python data science libraries. We evaluate the efficiency and pass@1 of LLM-generated code for the DS-1000 tasks using Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct, both in their original form and fine-tuned with EFFI-CODE. As shown in Table 5, fine-tuning with EFFI-CODE leads to modest improvements in code efficiency for both models. For Qwen2.5-Coder-7B-Instruct, the average execution time (ET) decreases by 0.8%, and the total memory usage (TMU) decreases by 0.9% after fine-tuning, while the pass@1 rate improves significantly from 17.00% to 35.70%. For deepseek-coder-6.7b-instruct, fine-tuning results in a 4.5% reduction in ET and a 5.6% reduction in TMU, with the memory usage (MU) and normalized memory usage (NMU) also decreasing by 1.9% and 2.0%, respectively, although the pass@1 rate remains the same at 37.50%. While the improvements in code efficiency for the DS-1000 benchmark are less pronounced compared to other benchmarks, the results still demonstrate that fine-tuning with EFFI-CODE can enhance the efficiency of LLM-generated code for data science tasks.

4.3 Different Programming Language

In addition to evaluating the efficiency and pass@1 of LLM-generated code for Python tasks, we also conduct experiments on the HumanEval-X dataset, where we focus on the C++ and Java subsets, to measure the performance of EFFI-CODE fine-tuned LLMs in a different-language setting. As shown in Table 6, both Qwen2.5-Coder-7B-Instruct and deepseek-coder-6.7b-instruct demonstrate significant improvements in code efficiency and pass@1 rates after fine-tuning with EFFI-CODE for C++ tasks. Qwen2.5-Coder-7B-Instruct achieves a 37.1% reduction in average execution time (ET), a 42.6% reduction in memory usage (MU), and a 53.8% reduction in total memory usage (TMU), with the pass@1 rate improving from 7.32% to 48.17%. Similarly, deepseek-coder-6.7b-instruct exhibits a 45.2% reduction in ET, a 50.0% reduction in MU, and a 40.0% reduction in TMU, with the pass@1 rate increasing from 14.63% to 40.24%. For Java tasks, deepseek-coder-6.7b-instruct achieves a 36.5% reduction in ET, a 50.5% reduction in MU, and a 30.4% reduction in TMU after fine-tuning, with the pass@1 rate improving from 14.63% to 57.93%, matching the performance of the fine-tuned Qwen2.5-Coder-7B-Instruct. These results demonstrate that fine-tuning with EFFI-CODE can significantly enhance the efficiency and correctness of LLM-generated code across multiple programming languages.

4.4 Comparison with Baselines

To further demonstrate the efficiency of the code generated by EFFI-CODE fine-tuned LLMs, we compare the performance of CodeLlama-7b-hf fine-tuned using EFFI-CODE with two baselines: PIE Shypula et al. [2024] and Mercury Du et al. [2024]. The evaluation results on the HumanEvalPlus dataset are presented in Table 7. We can observe that for the correct tasks addressed by both PIE and EFFI-CODE, PIE requires 0.30 (s) on average to address each task, which is a 23.1% reduction in average execution time compared to the original CodeLlama-7b-hf generated code. However, the EFFI-CODE fine-tuned CodeLlama-7b-hf reduces the average execution time by 46.2%, requiring only 0.21 (s) on average to address each correct task. Moreover, the EFFI-CODE fine-tuned model achieves a 46.9% reduction in NET, a 43.9% reduction in TMU, and a 43.2% reduction in NTMU compared to PIE. The pass@1 also improves from 19.5% for PIE to 31.1% for the EFFI-CODE fine-tuned model. Similarly, when compared to Mercury, the EFFI-CODE fine-tuned CodeLlama-7b-hf demonstrates a 46.2% reduction in average execution time, a 47.9% reduction in NET, a 45.6% reduction in TMU, and a 46.4% reduction in NTMU. The pass@1 rate improves significantly from 9.1% for Mercury to 31.1% for the EFFI-CODE fine-tuned model, with both methods having an overlapped percentage of 4.3%. The substantial improvements in efficiency and correctness achieved by the EFFI-CODE fine-tuned model demonstrate the effectiveness of this approach in optimizing LLM-generated code for practical applications.

5 Conclusion

In this paper, our research addresses a critical gap in the efficiency of code generated by LLMs by introducing the EFFI-CODE dataset, designed to enhance both the correctness and execution efficiency of LLM-generated code via fine-tuning. Through meticulous aggregation, preprocessing, and iterative optimization, we provide a robust resource that significantly boosts the performance of open-source LLMs like DeepSeek-Coder and Qwen. Our experiments reveal substantial improvements, with notable increases in pass rates and decreases in execution time, underscoring the potential of EFFI-CODE to advance the state of code generation in resource-constrained environments. By open-sourcing our model weights, training data, and source code, we aim to foster further research and innovation in this vital area of AI development tools.

References

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. AVATAR: A parallel corpus for java-python program translation. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 2268–2281. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-ACL.143. URL https://doi.org/10.18653/v1/2023.findings-acl.143.

Toufique Ahmed and Premkumar T. Devanbu. Few-shot training llms for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 177:1–177:5. ACM, 2022. doi: 10.1145/3551349.3559555. URL https://doi.org/10.1145/3551349.3559555.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023. doi: 10.48550/ARXIV.2301.03988. URL https://doi.org/10.48550/arXiv.2301.03988.

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023. doi: 10.48550/ARXIV.2312.11805. URL https://doi.org/10.48550/arXiv.2312.11805.

Anthropic. Introducing the next generation of claude, 2024. URL https://www.anthropic.com/news/claude-3-family.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

BigCode. Self-oss-instruct-sc2-exec-filter-50k. https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k, 2023.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating

large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *CoRR*, abs/2304.05128, 2023. doi: 10.48550/ARXIV.2304.05128. URL https://doi.org/10.48550/arXiv.2304.05128.

Cognitive Computations. Dolphin coder. https://huggingface.co/datasets/cognitivecomputations/dolphin-coder, 2023.

Together Computer. Glaive-code-assistant. https://huggingface.co/datasets/Together/Glaive-Code-Assistant, 2023.

Jianbo Dai, Jianqiao Lu, Yunlong Feng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. MHPP: exploring the capabilities and limitations of language models beyond basic code generation. *CoRR*, abs/2405.11430, 2024. doi: 10.48550/ARXIV.2405.11430. URL https://doi.org/10.48550/arXiv.2405.11430.

DeepSeekAI. Deepseek coder: Let the code write itself, 2023. URL https://deepseekcoder.github.io/.

Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *CoRR*, abs/2304.02014, 2023. doi: 10.48550/ARXIV.2304.02014. URL https://doi.org/10.48550/arXiv.2304.02014.

Mingzhe Du, Anh Tuan Luu, Bin Ji, and See-Kiong Ng. Mercury: An efficiency benchmark for LLM code synthesis. *CoRR*, abs/2402.07844, 2024. doi: 10.48550/ARXIV.2402.07844. URL https://doi.org/10.48550/arXiv.2402.07844.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=hQwb-lbM6EL.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *CoRR*, abs/2306.11644, 2023. doi: 10.48550/ARXIV.2306.11644. URL https://doi.org/10.48550/arXiv.2306.11644.

Md. Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. Fixeval: Execution-based evaluation of program fixes for competitive programming problems. *CoRR*, abs/2206.07796, 2022. doi: 10.48550/ARXIV.2206.07796. URL https://doi.org/10.48550/arXiv.2206.07796.

Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md. Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. Codesc: A large code-description parallel dataset. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 210–218. Association for Computational Linguistics, 2021. doi: 10.18653/V1/2021.FINDINGS-ACL.18. URL https://doi.org/10.18653/v1/2021.findings-acl.18.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Jie M. Zhang, Heming Cui, and Zhijiang Guo. SOAP: enhancing efficiency of generated code via self-optimization. *CoRR*, abs/2405.15189, 2024a. doi: 10.48550/ARXIV.2405.15189. URL https://doi.org/10.48550/arXiv.2405.15189.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. Effibench: Benchmarking the efficiency of automatically generated code. *arXiv preprint arXiv:2402.02037*, 2024b.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. 2024. URL https://api.semanticscholar.org/CorpusID:272707390.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1430–1442. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00125. URL https://doi.org/10.1109/ICSE48619.2023.00125.

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

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 919–931. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00085. URL https://doi.org/10.1109/ICSE48619.2023.00085.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *CoRR*, abs/2305.06161, 2023a. doi: 10.48550/ARXIV.2305.06161. URL https://doi.org/10.48550/arXiv.2305.06161.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need II: phi-1.5 technical report. *CoRR*, abs/2309.05463, 2023b. doi: 10.48550/ARXIV.2309.05463. URL https://doi.org/10.48550/arXiv.2309.05463.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022. doi: 10.48550/ARXIV.2203.07814. URL https://doi.org/10.48550/arXiv.2203.07814.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

MAP. Codefeedback-filtered-instruction. https://huggingface.co/datasets/m-a-p/CodeFeedback-Filtered-Instruction, 2023.

Meta. Introducing meta llama 3: The most capable openly available llm to date, 2024. URL https://ai.meta.com/blog/meta-llama-3/.

Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2241–2252. ACM, 2022. doi: 10.1145/3510003.3510124. URL https://doi.org/10.1145/3510003.3510124.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=mw1PWNSWZP.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=iaYcJKpY2B_.

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

OpenAI. GPT-4 Technical Report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/arXiv.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

Jieke Shi, Zhou Yang, and David Lo. Efficient and green large language models for software engineering: Vision and the road ahead. *arXiv preprint arXiv:2404.04566*, 2024.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana

Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. doi: 10.48550/ARXIV.2307.09288. URL https://doi.org/10.48550/arXiv.2307.09288.

ISE UIUC. Magicoder-evol-instruct-110k. https://huggingface.co/datasets/ise-uiuc/Magicoder-Evol-Instruct-110K, 2023a.

ISE UIUC. Magicoder-oss-instruct-75k. https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K, 2023b.

Vezora. Tested-143k-python-alpaca. https://huggingface.co/datasets/Vezora/Tested-143k-Python-Alpaca, 2023.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.ACL-LONG.754. URL https://doi.org/10.18653/v1/2023.acl-long.754.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021. doi: 10.18653/V1/2021.EMNLP-MAIN.685. URL https://doi.org/10.18653/v1/2021.emnlp-main.685.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=gEZrGCozdqR.

Jiayi Wei, Greg Durrett, and Isil Dillig. Typet5: Seq2seq type inference using static analysis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=4TyNEhI2GdN.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*, 2024a.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=XUeoOBid3x.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint arXiv:2403.19114*, 2024.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=CfXh93NDgH.

Chenyang Zhao, Xueying Jia, Vijay Viswanathan, Tongshuang Wu, and Graham Neubig. Self-guide: Better task-specific instruction following via self-synthetic finetuning. *arXiv preprint arXiv:2407.12874*, 2024.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

A Appendix

A.1 Broader Impacts

This paper presents work aimed at advancing the field of Machine Learning by improving the efficiency and correctness of code generated by LLMs. The societal benefits of this research include:

- Sustainability: By reducing computational resource consumption (e.g., energy and memory usage), our work aligns with global efforts to mitigate the environmental impact of large-scale computing. Efficient code generation could lower operational costs and energy demands in industries reliant on software development.
- Resource-Constrained Environments: Enhanced efficiency enables broader adoption of LLM-generated code in mobile, embedded, or edge computing systems, where energy and processing power are limited.
- Research Advancement: Open-sourcing our dataset and models fosters transparency and accelerates research into sustainable AI systems, encouraging further innovations in code optimization.

Ethical Considerations:

- Data Provenance: The dataset is aggregated from publicly available open-source repositories on Hugging Face, ensuring compliance with licensing terms. However, future work should continue to prioritize responsible data curation practices.
- Bias and Generalization: While our framework supports multilingual adaptability, biases in the source code (e.g., language-specific optimizations or cultural coding norms) may inadvertently propagate. Mitigating such biases requires careful dataset design and validation.
- Developer Dependency: Widespread adoption of optimized LLM-generated code could influence coding practices. Ensuring human developers retain critical problem-solving skills remains important.

Overall, this work aims to address a critical gap in LLM-generated code efficiency while maintaining correctness. We encourage future research to explore trade-offs between efficiency, maintainability, and fairness in automated code generation.

A.2 Prompt Template

Please continue to complete the function. You are not allowed to modify the given code and do the completion only. Please return all completed functions in a code block. Here is the given code to complete:
```python
{{Prompt}}
```

A.3 Efficiency Metrics

Execution Time (ET) Execution time (ET) measures the average time taken for code execution. Mathematically, ET is defined as:

$$ET = \frac{1}{N}\sum^{N} T_{\text{code}}$$

where ET is the execution time metric, T_{code} is the execution time of the code (with all the test cases), and N is the number of codes generated by code generation models used for evaluation.

Normalized Execution Time (NET) Normalized Execution Time (NET) measures the execution time required by generated code relative to that of a canonical solution. We define NET as:

$$NET = \frac{1}{N}\sum^{N} \frac{T_{\text{code}}}{T_{\text{canonical}}}$$

where T_{code} is the execution time of the generated code and $T_{\text{canonical}}$ is the execution time of the canonical solution. A NET value greater than 1 indicates that the generated code is slower than the canonical solution, while a value less than 1 suggests the generated code is faster.

Max Memory Usage (MU) Max Memory Usage (MU) measures the average max memory consumption during code execution. Mathematically, MU is defined as:

$$MU = \frac{1}{N} \sum^{N} M_{\text{code}}$$

where MU is the memory usage metric, M_{code} is the max memory consumption of the generated code among all the test cases, and N is the number of code instances generated by code generation models used for evaluation. This metric is critical to assess the resource efficiency of generated code, particularly in environments with limited maximum memory capacity.

Normalized Max Memory Usage (NMU) Normalized Max Memory Usage (NMU) quantifies how the max memory efficiency of the generated code compares to the canonical solution. We define NMU as:

$$NMU = \frac{1}{N} \sum^{N} \frac{M_{\text{code}}}{M_{\text{canonical}}}$$

where NMU is the normalized max memory usage metric, M_{code} is the max memory usage of the generated code, and $M_{\text{canonical}}$ is the max memory usage of the canonical solution. An NMU value less than 1 indicates that the generated code is more memory-efficient than the canonical solution, whereas a value greater than 1 suggests it is less efficient in terms of memory usage. This metric provides a relative measure of the memory optimization in the generated code in comparison to a standard baseline.

Total Memory Usage (TMU) Total Memory Usage (TMU) assesses the efficiency of memory usage throughout the execution of code, taking into account both the magnitude and duration of memory utilization. To calculate TMU, first, monitor and record the memory usage at discrete time intervals during the execution, resulting in a memory usage profile $M(t)$, where t represents time. Then, compute the area under the curve of $M(t)$ over the total execution time, T_{total}, using numerical integration methods such as the trapezoidal rule:

$$TMU = \frac{1}{N} \sum^{N} \int_{0}^{T_{\text{total}}} M(t)\, dt$$

A lower TMU value indicates higher memory efficiency, reflecting an optimized balance between the amount of memory used and the duration of its usage.

Normalized Total Memory Usage (NTMU) The Normalized Total Memory Usage (NTMU) offers a comparison of the dynamic memory efficiency between the generated code and the canonical solution. To determine NTMU, calculate the TMU for both the generated code and the canonical solution. Normalize the TMU of the generated code by dividing it by the TMU of the canonical solution:

$$NTMU = \frac{1}{N} \sum^{N} \frac{TMU_{\text{code}}}{TMU_{\text{canonical}}}$$

where TMU_{code} is the TMU of the generated code and $TMU_{\text{canonical}}$ is the TMU of the canonical solution. An NTMU value less than 1 signifies that the generated code manages dynamic memory more efficiently compared to the canonical solution, while a value greater than 1 indicates less efficient management of dynamic memory. This metric provides insight into the relative use of dynamic memory of generated code compared to an established benchmark.

A.4 Robustness of Overhead Results

The overhead results would be affected by the local environments, which causes that the results of Effi-Code fine-tuned LLMs may not able to represent the results of the efficiency profiling in different environments. To address this issue, we have conducted additional experiments and provided more robust evaluation results.

Setup	ET	NET	MU	NMU	TMU	NTMU
Python 3.11.10 - Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz						
Qwen2.5-Coder-7B	0.59	1.95	61.95	0.99	24.29	1.83
+Effi-Code	0.40	1.01	61.96	0.99	18.74	1.02
Python 3.11.10 - Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz						
Qwen2.5-Coder-7B	0.28	1.63	36.15	1.00	20.01	1.88
+ SFT	0.25	1.38	36.52	1.01	19.85	1.56
Python 3.11.10 - Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz						
Qwen2.5-Coder-7B	0.35	1.45	36.14	1.00	24.28	1.63
+ SFT	0.22	1.01	36.51	1.01	15.26	1.09
Python 3.11.4 - Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz						
Qwen2.5-Coder-7B	0.67	1.16	61.43	1.00	40.01	1.22
+Effi-Code	0.58	1.02	60.77	0.97	32.50	1.03
Python 3.11.0 - Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz						
Qwen2.5-Coder-7B	0.28	1.64	34.55	1.00	19.39	1.87
+ SFT	0.25	1.39	34.90	1.02	20.03	1.59
Python 3.9.0 - Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz						
Qwen2.5-Coder-7B	0.30	1.60	34.26	1.01	21.02	2.10
+Effi-Code	0.24	1.20	34.52	1.02	19.84	1.32
Python 3.10.0 - Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz						
Qwen2.5-Coder-7B	0.29	1.63	33.26	1.01	20.32	2.16
+ SFT	0.26	1.43	33.50	1.02	19.53	1.61

Table 8: Evaluation results of Effi-Code's effectiveness on different software-hardware setups.

Firstly, we have evaluated the effectiveness of Effi-Code on seven different software-hardware setups, as shown in Rebuttal Table 2. The results demonstrate that Effi-Code fine-tuned LLMs achieve higher efficiency than the original LLMs across all setups. For example, in the environment of Python 3.11.10 - Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz, the average execution time decreases from 0.59s to 0.40s when using Effi-Code to fine-tune Qwen2.5-Coder-7B, reducing the average execution time by 32%.

Secondly, we clarify that for the same setup, where we evaluate the efficiency of LLM-generated code several times, the efficiency results are consistent. As shown in Paper Table 8, where we execute the LLM-generated code five times, the standard deviation of execution time (ET) is 0.00548 (s), indicating that the evaluation results are consistent and reliable for a given setup.

Finally, our evaluation setup follows the practices established in recent works on benchmarking the efficiency of automatically generated code, such as Mercury Du et al. [2024], Effibench Huang et al. [2024b], and SOAP Huang et al. [2024a]. By adhering to these benchmarks, we ensure that our evaluation is in line with the current standards in the field.x

A.5 Test case augmentation

Some of the candidate tasks we collected do not have test cases. To address this, we use GPT-3.5-turbo to construct test cases by feeding the task description and source code into the model and requiring it to generate test cases for our experiments. After that, we analyze whether each test case generated by GPT-3.5-turbo is correct and then filter out incorrect test cases and tasks that do not have the correct test cases. To determine the correctness of the test cases generated by GPT-3.5-turbo, we execute each test case individually with the initial solution for each task in our collected candidate tasks. We check whether any errors are raised during the execution of each test case with the initial solution. In other words, we verify if the test case passes the initial solution. We treat the test cases that pass

the initial solution as correct. On the other hand, test cases that do not pass the initial solution are filtered out. By using the initial solution as a reference, we can effectively assess the correctness of the generated test cases and ensure that only valid test cases are retained for further analysis.

A.6 Case Study

To illustrate how the source code generated by EFFI-CODE fine-tuned LLM is more efficient than the source code generated by the LLM without fine-tuning on EFFI-CODE, we provide an example in Figure 5. We can observe that the code generated by Qwen2.5-Coder-7B requires 9.89 (s) to execute all unit tests, while the code generated by EFFI-CODE fine-tuned Qwen2.5-Coder-7B with SFT only requires 0.14 (s) to execute. The key reason is that the code generated by Qwen2.5-Coder-7B requires significantly more recursive calls, as it lacks optimized pruning strategies such as breaking early in redundant paths. This inefficiency leads to a much larger number of computations, ultimately resulting in the observed longer execution time. The code generated by EFFI-CODE fine-tuned Qwen2.5-Coder-7B, on the other hand, incorporates smart optimizations, such as terminating recursion early when certain conditions are met, thereby reducing the overall time complexity.

Task Description

Problem: You are given an integer array `cookies`, where `cookies[i]` denotes the number of cookies in the `ith` bag. You are also given an integer `k` that denotes the number of children to distribute `all` the bags of cookies to. All the cookies in the same bag must go to the same child and cannot be split up. The `unfairness` of a distribution is defined as the `maximum total` cookies obtained by a single child in the distribution. Return the `minimum` unfairness of all distributions.
solution = Solution() assert solution.distributeCookies([8, 15, 10, 20, 8], 2) == 31 assert solution.distributeCookies([6, 1, 3, 2, 2, 4, 1, 2], 3) == 7

```
solution = Solution()
assert solution.distributeCookies([8, 15, 10, 20, 8], 2) == 31
assert solution.distributeCookies([6, 1, 3, 2, 2, 4, 1, 2], 3) == 7
```

Qwen2.5-Coder-7B

```
Execution Time: 9.89 (s)
Max Memory Peak: 24.39 (MB)
Memory Usage: 240.47 (MB*s)
class Solution:
    def distributeCookies(self, cookies, k):
        min_unfairness = float('inf')
        distribution = [0] * k
        def distribute(i):
            nonlocal min_unfairness
            if i == len(cookies):
                min_unfairness = min(min_unfairness
                max(distribution))
                return
            for j in range(k):
                distribution[j] += cookies[i]
                distribute(i + 1)
                distribution[j] -= cookies[i]
        distribute(0)
        return min_unfairness
```

Qwen2.5-Coder-7B SFT with EFFI-CODE

```
Execution Time: 0.14 (s)
Max Memory Peak: 24.39 (MB)
Memory Usage: 2.47 (MB*s)
class Solution:
    def distributeCookies(self, cookies, k):
        def backtrack(i):
            nonlocal ans
            if i == len(cookies):
                ans = min(ans, max(children))
                return
            for j in range(k):
                children[j] += cookies[i]
                backtrack(i + 1)
                children[j] -= cookies[i]
                if children[j] == 0:
                    break
        children = [0] * k
        ans = float('inf')
        backtrack(0)
        return ans
```

Figure 5: A case illustration for the task with code generated by Qwen2.5-Coder-7B and EFFI-CODE fine-tuned Qwen2.5-Coder-7B in EffiBench `problem_idx=2305`.

A.7 Randomness

To ensure reliable model performance, we also account for variability in system conditions. Metrics like Execution Time (ET), Max Memory Usage (MU), and Total Memory Usage (TMU) might fluctuate due to factors like server workload and hardware availability, introducing noise that affects performance measurements. To demonstrate whether our results are affected by such randomness, we provide five results at different times with the mean and std for Qwen2.5-Coder-7B fine-tuned

Table 9: Code efficiency and pass@1 of Qwen2.5-Coder-7B with EFFI-CODE with the five times execution on EffiBench.

Model	**ET (s)** ↓	**NET** ↓	**MU (Mb)** ↓	**NMU** ↓	**TMU (Mb*s)** ↓	**NTMU** ↓
Random Execution 1	0.17	1.30	32.71	1.03	8.31	2.23
Random Execution 2	0.17	1.31	32.93	1.04	8.35	2.28
Random Execution 3	0.17	1.30	32.71	1.03	8.23	2.22
Random Execution 4	0.17	1.30	32.84	1.04	8.30	2.25
Random Execution 5	0.17	1.30	32.88	1.04	8.28	2.27
mean	0.17	1.302	32.814	1.037	8.293	2.249
std	0.0	0.003	0.09	0.003	0.038	0.023

with EFFI-CODE in Table 9. We can observe that the results are robust as the std of the five execution times is very low for all metrics. For example, the std of ET for the five executions is 0.00.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: **[TODO]**

   Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include experiments.

- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.

- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: **[TODO]**

   Justification: **[TODO]**

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: **[TODO]**

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: **[TODO]**

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: **[TODO]**

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: **[TODO]**

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: **[TODO]**

    Justification: **[TODO]**

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: **[TODO]**

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: **[TODO]**

    Justification: **[TODO]**

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.