On Evaluating the Efficiency of Source Code Generated by LLMs

Changan Niu
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
niu.ca@outlook.com

Ting Zhang
School of Computing and Information Systems
Singapore Management University
Singapore
tingzhang.2019@phdcs.smu.edu.sg

Chuanyi Li
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
lcy@nju.edu.cn

Bin Luo
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
luobin@nju.edu.cn

Vincent Ng
Human Language Technology Research Institute
University of Texas at Dallas
Richardson, Texas, USA
vince@hlt.utdallas.edu

ABSTRACT

Recent years have seen the remarkable capabilities of large language models (LLMs) for code generation. Different from existing work that evaluate the correctness of the code generated by LLMs, we propose to further evaluate its efficiency. More efficient code can lead to higher performance and execution efficiency of programs and software completed by LLM-assisted programming. First, we evaluate the efficiency of the code generated by LLMs on two benchmarks, HumanEval and MBPP. Then, we choose a set of programming problems from the online judge platform LeetCode to conduct a more difficult evaluation. Finally, we explore several prompts that would enable LLMs to generate more efficient code.

ACM Reference Format:
Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. In *AI Foundation Models and Software Engineering (FORGE '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650105.3652295

1 INTRODUCTION

With the advent of large language models (LLMs) and abundant source code data, program synthesis has entered a new era, aiming to automatically generate correct and compliant code from natural language descriptions. Extensive work has demonstrated the ability of LLMs to excel at generating well-compliant code [9, 15, 27]. OpenAI's GPT-4 achieves 67.0% Pass@1 on HumanEval [22], a key benchmark for measuring functional correctness by given the natural language description [10]. Other open source LLMs like Code Llama [23] and WizardCoder [18] also demonstrate impressive results, with Code Llama reaching up to 53% and 55% on HumanEval and MBPP [6], and WizardCoder achieving 57.3% and 51.8% on the same benchmarks with just 7B parameters.

Prompt of HumanEval/94

```
def skjkasdkd(lst):
    """You are given a list of integers.
    You need to find the largest prime value and return the sum of its
        digits.
    ...
```

Solution by GPT-3.5-turbo-1106

```
...
for i in range(2, int(n**0.5) + 1):
    if n % i == 0:
        return False
...
```

Solution by Code Llama 34B

```
...
for j in range(2, i):
    if (i % j == 0):
        prime = False
...
```

Figure 1: Code snippets extracted from the LLM-generated code for HumanEval.

Given LLM's impressive performance in code generation, a number of LLM-based programming assistance tools have emerged, such as GitHub Copilot [13], and JetBrains' AI Assistant [16]. These tools can offer intelligent code suggestions that automatically complete the code based on the context and the programmer's intent, thus making coding faster and speeding up the development process.

However, the efficiency of the generated code is overlooked. In Figure 1, GPT-3.5 and Code Llama's solutions on the HumanEval/94 example both yield correct code. However, GPT-3.5's solution exhibits higher running efficiency due to its $O(\sqrt{n})$ complexity compared to Code Llama's $O(n)$ complexity for determining prime numbers. This highlights the potential differences in execution efficiency among LLM-generated code. Recommending more efficient code not only enhances program/software performance but also increase the probability of code acceptance by developers, reducing the need for further optimization and boosting development productivity. Therefore, investigating and discussing the efficiency of LLM-generated code is essential, assuming the functional correctness of the code is ensured.

Consequently, in this paper, we propose to conduct an empirical study on the efficiency of LLM-generated code by investigating the following research questions (RQs):

FORGE '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0609-7/24/04…$15.00
https://doi.org/10.1145/3650105.3652295

RQ1: How efficient is the code generated by LLMs?
RQ2: How to prompt LLMs for more efficient code?

For RQ1, we measure and compare the execution time (we abbreviate this to "runtime" in this paper) of the code generated by LLMs first on two entry-level programming benchmarks, HumanEval and MBPP and then on a benchmark containing more complex problems. For RQ2, we try various prompts to explore how to make LLM generate code that executes more efficiently. Results show that simple prompts enhance efficiency for basic problems, while complex problems benefit from a chain-of-thought prompt.

This paper makes three contributions: (1) evaluate the efficiency of the code generated by LLMs. The results may guide practitioners in choosing the most suitable model based on their specific requirements, (2) propose a LeetCode-based benchmark which provides a reference point for comparing the correctness and efficiency of more complex code, (3) investigate to prompt LLM for generating more efficient code, which could directly benefit developers and organizations using these models in various applications. We also make code, data and other artifacts available online [1].

2 APPROACH AND EXPERIMENTS

In this section, we describe how we design and conduct experiments to investigate and answer two RQs.

2.1 RQ1: Efficiency of LLM-generated Code

2.1.1 Datasets. We evaluate the efficiency of LLM-generated code using two entry-level programming benchmarks, HumanEval and MBPP, and a benchmark containing more complex problems.

HumanEval and MBPP. HumanEval is used to measure functional correctness for synthesizing programs from docstrings. It consists of 164 original programming problems in Python, assessing language comprehension, algorithms, and simple mathematics, with some comparable to simple software interview questions. MBPP consists of a set of crowd-sourced Python programming problems, designed to be solvable by entry-level programmers, covering programming fundamentals, standard library functionality, etc.

LeetCodeEval. LeetCode [3] is a popular online judge platform that offers a wide range of problems. For each problem, LeetCode has a huge number of test cases covering a whole range of input sizes and scenarios. For accepted code, LeetCode will also give its runtime and the percentage of total code that it beats. Therefore, we propose to use LeetCode problems and the LeetCode platform to evaluate the correctness and efficiency of LLM-generated code.

In order to avoid data leakage, we select only problems from May 2023 and later (this is the latest GPT-4 knowledge cut-off). Besides, we filter out problems with images in the description and those have more downvotes than upvotes. Then, we divide the problems into three subsets according to difficulty levels officially given by LeetCode: easy, medium and hard. For each problem, we collect its URL, title, description, examples, constraints, code templates, etc. Ultimately, we build LeetCodeEval, a dataset for evaluating code correctness and efficiency based on the LeetCode platform, which consists of 44, 85 and 33 easy, medium and hard problems.

The statistics of datasets are presented in Table 1.

2.1.2 Study Design. HumanEval and MBPP. First, we use the source code provided by Liu et al. [17] to let the LLM generate responses for each problem by inputting the unfinished code prompt. Since only correct code can be used in our comparison, we make the LLM generate k responses for each problem to improve the possibility of collecting correct code. Then, we execute each code on corresponding test cases to determine if it is correct. If any one of the k codes generated by a LLM passes all the test cases, we consider the LLM to have passed the problem, and take the first passing code for efficiency evaluation in the next step. Otherwise, the LLM is considered to have failed on that problem. Next, we measure the runtime of each selected correct code. However, executing the code on real hardware directly will introduce a lot of noise due to machine loads, configurations, etc. [19]. Therefore, we utilize the gem5 CPU simulator [8], which is the mostly used golden standard in both academia and industry and is able to ensure the evaluation progress reliable and reproducible [4, 19]. To ensure the evaluation statistically significant, we repeat the execution of each piece of code for 10 times and take the average runtime as the final result.

Table 1: Dataset statistics. The prompt length is the GPT-2 tokenizer length.

Item	HumanEval	MBPP	LeetCodeEval		
			Easy	Medium	Hard
# of Problems	164	399	48	85	33
Mean Prompt Length	170.33	52.07	540.27	623.04	720.00
Median Prompt Length	145.5	47	530	583	665
Mean # of Test Cases	9.57	3.10	1840.10	1286.2	1036.64
Median # of Test Cases	7	3	906	785	774

````
Please solve the following programming problem entitled "{title}"
in C++, the problem is described below:

{description}

{examples}

{constraints}

Please use the following code template:
```cpp
{code_template}
```
````

Figure 2: The prompt template for LeetCodeEval.

LeetCodeEval. We focus on one of the performance-oriented languages, i.e., C++. For each problem in LeetCodeEval, we first prompt LLMs to generate 3 different C++ codes. Figure 2 present the prompt template obtained by asking ChatGPT. For each generated code, we submit it to the LeetCode platform and get its correctness and runtime (if accepted). Acceptance of any of the 3 codes is considered as LLM acceptance and the runtime of the first accepted code is recorded, otherwise it is considered as failure. We repeat the submission of each piece of code for 3 times and record the average results.

2.1.3 Models. We select commercial and open source LLMs that achieve the SOTA performance on HumanEval and MBPP:

GPT-3.5 and GPT-4. OpenAI's GPT-3.5 and GPT-4 can be seen as the most powerful LLM. We use two models by using the OpenAI API with model ids `gpt-3.5-turbo-1106` and `gpt-4-1106-preview`.

Phi-2. Phi-2 [21] is a 2.7B-parameter model that demonstrates outstanding reasoning and language understanding capabilities, showcasing SOTA performance among LLMs smaller than 13B.

Code Llama. Code Llama [23] is built on top of Llama 2 [25] and is fine-tuned for generating and discussing code. The 7B version is shown to outperform Llama 2 70B on both HumanEval and MBPP.

WizardCoder. WizardCoder [18] empowers code LLMs with complex instruction fine-tuning and outperforms the largest closed LLMs, Anthropic's Claude [5] and Google's Bard [14], on HumanEval.

DeepSeek Coder. DeepSeek Coder [7, 11] 33B version is able to outperform GPT-3.5 on HumanEval and achieve comparable results with GPT-3.5 on MBPP after instruct tuning. We choose the 33B version of DeepSeek Coder before and after instruct tuning for the experiment, denoted as the "base" and "instruct", respectively.

For LeetCodeEval, since it requires a chat/instruction model, we choose **GPT-4**, **GPT-3.5** and **DeepSeek Coder 33B Instruct**, which perform the best on LeetCode problems in our pre-experiments.

2.1.4 Metrics. We report average normalized runtime and Pass@10. Pass@10 metric is the probability that at least one of the top 10-generated code samples for a problem passes all test cases.

Since there is no runtime on failed problem, we compute runtime metric only for problems where all LLMs pass. For each such problem, we count the runtime of each code on all test cases. Let $t(M_j)^i = \{t(M_j)^i_1, ..., t(M_j)^i_n\}$ denotes the runtime of the code c^j generated by LLM M_j on test cases $tc^i = \{tc^i_1, ...tc^i_n\}$, where n is the number of test cases of problem p^i. Then, following the practice of online programming websites such as LeetCode [3] and Codeforces [2], we take the longest of these runtimes, i.e., $max(t^i)$, as the final runtime of the LLM M_j on the problem p^i.

Nevertheless, there are order of magnitude differences in the runtime of LLM on different problems, we normalize all the runtimes of all LLMs on each problem. Concretely, for the problem p^i, we let $t(M)^i = \{t(M_1)^i, ..., t(M_l)^i\}$ be the runtime of LLMs $M = \{m_1, ..., m_l\}$ on problem p^i, where l is the number of LLMs. Then the normalized runtime of LLMs on problem p^i is calculated as $normalize(t(M)^i) = \{\frac{t(M_1)^i}{sum(t(M)^i)}, ..., \frac{t(M_l)^i}{sum(t(M)^i)}\}$, where $sum(t(M)^i)$ denotes the summarization of all elements in $t(M)^i$. Then, the average normalized runtime of the LLM M_j is denoted as $\frac{\sum_{i=1}^{o} normalize(t(M_j)^i)}{o}$, where o is the number of problems that all LLMs pass, $normalize(t(M_j)^i)$ is the normalized runtime of the LLM M_j on the problem p^i.

Besides, we also adopt average percentage beats for LeetCodeEval, i.e., the average of the percentage of each accepted code that beats the other users.

2.1.5 Results. Table 2 and Table 3 shows the results of LLM on HumanEval and MBPP, and LeetCodeEval, respectively. Note that the average normalized runtime is computed based only on the programming problems that all LLMs pass, and there are 70 and 242 problems that all LLMs pass in HumanEval and MBPP, 24, 3 and 0 on easy, medium and hard subsets of LeetCodeEval, respectively. So, we cannot compare models' performances on the hard subset, and for medium subset, we find that 12 medium problems passed by both GPT-4 and DeepSeek Coder, so we only compare and report the two models on medium set.

Table 2: Results on HumanEval and MBPP.

LLM	Version	HumanEval		MBPP	
		Runtime	Pass@10	Runtime	Pass@10
GPT-4	N/A	8.61	98.2	9.14	94.2
GPT-3.5	N/A	8.35	87.2	8.86	88.7
Phi-2	2.7B	8.78	62.8	8.98	74.7
Code Llama	7B	9.95	68.9	9.58	81.0
	13B	9.87	79.3	9.61	83.0
	34B	9.93	80.5	9.54	85.0
WizardCoder	7B	9.35	67.7	8.54	74.4
	13B	9.18	75.0	8.83	81.5
	34B	9.04	83.5	8.60	85.5
DeepSeek Coder	33B Base	9.40	79.9	9.42	82.0
	33B Instruct	7.54	93.9	8.93	90.0

Table 3: Results of LLMs on easy and medium subsets. GPT-3.5 is excluded on the Medium subset.

LLM	Easy		Medium	
	Runtime	%Beats	Runtime	%Beats
GPT-4	30.89	65.51	50.92	73.09
GPT-3.5	33.80	62.08	-	-
DeepSeek Coder	35.30	61.05	49.08	67.46

First, the ability to generate correct code is not positively correlated with the ability to generate efficient code. For example, the Pass@10 of GPT-4 has a clear advantage over GPT-3.5, but the code generated by the former is not as efficient as the latter on both HumanEval and MBPP. The same happens with Phi-2, which, despite having the lowest Pass@10, generates code with a lower runtime than most of the other models. **Second, larger number of parameters does not promise higher performance.** Code Llama and WizardCoder series demonstrate that increasing the number of parameters does not significantly affect the runtime of generated code across models of different sizes. This suggests that models of varying sizes share similar performance due to their reliance on the same training data. **Then, training strategy and data have an impact on the efficiency of the generated code.** For example, DeepSeek Coder 33B Instruct has a significant advantage over its Base version. Indeed the "Base" version is trained on code corpus by completion and fill-in-the-blank tasks, while the "Instruct" version is the result of further instruct-tuning of the "Base" version on the instruction data. **Last, LLM performs differently across benchmarks.** On HumanEval, DeepSeek Coder 33B Instruct has the lowest runtime, but on MBPP, the lowest model becomes the WizardCoder series. We argue that this is related to the data distribution of the model and the dataset. In addition, on LeetCodeEval, the code generated by GPT-4 has the highest efficiency on average. We believe this is due to more diverse test cases compared to HumanEval and MBPP. Comprehensive test cases on LeetCode can make the runtime benefits of code with real less complexity more significant, and thus more accurately reflect the efficiency.

2.2 RQ2: Prompting for More Efficient Code

We try three different prompts which are illustrated in Figure 3, where the last two prompts are introduced by Madaan et al. [19].

Prompt 1

```
User: {original_prompt}
Please make the code as time efficient as possible.
LLM: {fast_code}
```

Prompt 2

```
User: {original_prompt}
LLM: {slow_code}
User: Optimize the code and provide a more efficient version.
LLM: {fast_code}
```

Prompt 3

```
User: {original_prompt}
LLM: {slow_code}
User: Give a potential strategy improving the efficiency of the
code.
LLM: strategy
User: Now give the optimized version of the same code with the
strategy mentioned above.
LLM: {fast_code}
```

Figure 3: Three prompt methods.

Table 4: Speedup of three prompt methods.

Method	LLM	HumanEval	MBPP	LeetCodeEval	
				Easy	Medium
Prompt 1	GPT-4	1.06	1.04	1.13	1.07
	GPT-3.5	1.04	1.03	1.11	-
	DeepSeek Coder	1.00	1.01	1.03	1.02
Prompt 2	GPT-4	1.06	1.05	1.15	1.16
	GPT-3.5	1.03	1.03	1.15	-
	DeepSeek Coder	1.01	1.02	1.05	1.02
Prompt 3	GPT-4	1.05	1.04	1.18	1.18
	GPT-3.5	1.04	1.03	1.16	-
	DeepSeek Coder	1.01	1.00	1.05	1.01

Prompt 1 directly asks the LLM to generate the code as efficient as possible. Both prompt 2 and prompt 3 are chain-of-thought prompts. They first use the original prompt to make the model generate the original code. Then prompt 2 asks model to optimize it, while prompt 3 first has the model analyze optimization strategies before generating the optimized code.

We apply the three prompts on GPT-4, GPT-3.5 and DeepSeek Coder 33B Instruct. Note that here, same as in RQ2, both the LeetCodeEval hard subset and the GPT-3.5 on the LeetCodeEval medium subset are excluded from evaluation. For metrics, we choose the speedup rate, i.e., let t_o and t_n be the runtime of the original code and optimized code, respectively, then $speedup = \frac{t_o}{t_n}$. Following Madaan et al. [19], for cases where the optimized code fails or the runtime is higher, we make $speedup = 1$.

The overall results are shown in Table 4. **First, the prompt method generally works better on LeetCodeEval than on HumanEval and MBPP.** We believe there are two reasons: (1) HumanEval and MBPP problems have lower average difficulty and complexity than LeetCodeEval, resulting in a constrained optimization space and similar performance across prompt methods, and (2) the limited input size of the former prevents the reduction in algorithmic complexity from being evident in the runtime, whereas the more extensive test cases in LeetCodeEval magnify the performance of code with lower complexity. **Second, the three prompts have a larger gap on the medium subset of LeetCodeEval than easy subset.** This is because the simplicity of the easy subset allows the model to produce correct and efficient code simultaneously. However, the increased complexity in the medium subset, due to higher problem difficulty, hinders Prompt 1 from generating compliant code in a single step. Improved results are achieved by having the model first generate correct code and then analyze and optimize it step by step.

3 THREATS TO VALIDITY

Potential data leakage is a threat to construct validity because we can not know if the data used for evaluation is present in the training data of models. We mitigate this threat by selecting only LeetCode problems after April 2023, which is the latest knowledge cut-off of the GPT series, however, we are unable to get the data cut-offs for the other model. Threats to internal validity is related to the unstable runtime, we mitigate this by using the gem5 CPU simulator and running each evaluation process multiple times.

4 RELATED WORK

DeepDev-PERF [12], a deep learning-based approach to improve software performance for C# applications, can generate the same performance improvement suggestions as the developer patches in 53% of the cases. Madaan et al. [19] adapt LLMs to code optimization with respect to the runtime. They propose PIE, a dataset consists of C++ program pairs with runtime annotations, and evaluate different prompting and fine-tuning approaches for adapting LLMs to optimize programs. By allowing LLMs to iteratively provide self-feedback and refine their own outputs, Self-Refine [20] increases the LLM's performace on PIE dataset.

Rather than efficiency, Siddiq et al. [24] evaluate and improve the quality of the automatically generated code by LLMs w.r.t. the adherence to coding standards and presence of code smells and security smells. Yetiştiren et al. [26] assesses the code generation capabilities of several LLMs in terms of code quality metrics, such as code validity and maintainability.

5 CONCLUSION AND FUTURE WORK

This paper evaluates the efficiency of LLM-generated code, revealing that (1) the efficiency of LLM-generated code is independent of the model's performance on generating correct code and model size, and (2) step-by-step prompting could make LLM to generate more efficient code, especially on complex problems. Our study suggests a research avenue for improving LLMs in code efficiency, offering practical insights for model selection. Future work will focus on proposing a novel prompt method to enhance LLM-generated code efficiency.

ACKNOWLEDGMENTS

This research is supported by the Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming, CCF-Huawei Populus Grove Fund, NSF award 2034508. We also thank the reviewers for their helpful comments. Chuanyi Li is the corresponding author.

REFERENCES

[1] [n. d.]. https://github.com/NougatCA/EfficiencyEval.
[2] [n. d.]. Codeforces. https://codeforces.com/.
[3] [n. d.]. LeetCode. https://leetcode.com/.
[4] Ayaz Akram and Lina Sawalha. 2019. Validation of the gem5 simulator for x86 architectures. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 53–58.
[5] Anthropic. [n. d.]. Introducing Claude. https://www.anthropic.com/index/introducing-claude.
[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]
[7] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. *arXiv preprint arXiv:2401.02954* (2024).
[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.
[9] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=ktrw68Cmu9c
[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[11] DeepSeek. 2023. DeepSeek Coder: Let the Code Write Itself. https://github.com/deepseek-ai/DeepSeek-Coder.
[12] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. 2022. DeepDev-PERF: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 948–958.
[13] GitHub. [n. d.]. GitHub Copilot. https://github.com/features/copilot.
[14] Google. [n. d.]. Bard. https://bard.google.com/.
[15] Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, and Yunji Chen. 2023. ANPL: Towards Natural Programming with Interactive Decomposition. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=RTRS3ZTsSj
[16] JetBrains. [n. d.]. JetBrains AI. https://www.jetbrains.com/ai/.
[17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.
[18] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs.CL]
[19] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *International Conference on Learning Representations*.
[20] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651* (2023).
[21] Microsoft. 2023. Phi-2: The surprising power of small language models. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/.
[22] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[23] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
[24] Mohammed Latif Siddiq, Beatrice Casey, and Joanna Santos. 2023. A Lightweight Framework for High-Quality Code Generation. *arXiv preprint arXiv:2307.08220* (2023).
[25] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[26] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778* (2023).
[27] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic Reasoning with Language Models by Composing Decompositions. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=qd9qcbVAwQ