A Performance Study of LLM-Generated Code on Leetcode

Tristan Coignion, Clément Quinton, Romain Rouvoy

A Performance Study of LLM-Generated Code on Leetcode

Tristan Coignion
Univ. Lille, CNRS, Inria
France
tristan.coignion@inria.fr

Clément Quinton
Univ. Lille, CNRS, Inria
France
clement.quinton@inria.fr

Romain Rouvoy
Univ. Lille, CNRS, Inria
France
romain.rouvoy@inria.fr

ABSTRACT

This study evaluates the efficiency of code generation by *Large Language Models* (LLMs) and measures their performance against human-crafted solutions using a dataset from Leetcode. We compare 18 LLMs, considering factors such as model temperature and success rate, and their impact on code performance. This research introduces a novel method for measuring and comparing the speed of LLM-generated code, revealing that LLMs produce code with comparable performance, irrespective of the adopted LLM. We also find that LLMs are capable of generating code that is, on average, more efficient than the code written by humans. The paper further discusses the use of Leetcode as a benchmarking dataset, the limitations imposed by potential data contamination, and the platform's measurement reliability. We believe that our findings contribute to a better understanding of LLM capabilities in code generation and set the stage for future optimizations in the field.

ACM Reference Format:
Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A Performance Study of LLM-Generated Code on Leetcode. In *Proceedings of The 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large Language Models (LLMs) have recently increased in popularity, especially with the advent of ChatGPT [28]. While LLMs have been spreading over various application domains, such as text or image generation, certain types of LLMs are being developed solely for code-related purposes. These LLMs aim to assist the developers by saving time and effort through the generation of code, documentation, unit tests, etc. Many of these LLMs only come in a "raw" form, that is, they do not integrate themselves into the developer's coding process. These include models, such as CodeGen [27], StarCoder [21], WizardCoder [24], CodeT5 [38], and Incoder [14]. On the other hand, some LLMs are already seamlessly integrated into the developer's IDE as code assistants, like GitHub Copilot,[1] Amazon CodeWhisperer,[2] and Tabnine.[3]

[1]https://github.com/features/copilot
[2]https://aws.amazon.com/fr/codewhisperer/
[3]https://www.tabnine.com/

EASE 2024, 18–21 June, 2024, Salerno, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There has been a significant amount of work dedicated to comprehending how these LLMs perform in various situations and defining their limits. For instance, several works address the security of the code generated by such models [29, 30, 33] or the prevalence of bugs in the generations [20]. Many researchers are also investigating how developers interact with LLMs and how such models fit into the programming workflow [7, 30, 34] There is also a broad research effort to measure the actual efficiency of these LLMs by creating common benchmarks for comparison [5, 10, 17, 37, 43] or by actually measuring different qualities related to the generations, such as the success rate [42] or the robustness of the model regarding variations [13].

To the best of our knowledge, there is no research work evaluating the performance of the code generated by LLMs. Yet, having code that runs faster is an often sought-after characteristic of a program. Indeed, programming efficiency is paramount, especially when resources are scarce or programs are deployed on a large scale. In today's context where the energy consumption of software systems has become a major concern, improving software efficiency is particularly relevant since increasing the performance of a program can also lead to energy consumption reduction [2, 36].

The process of code optimization is lengthy and intricate, requiring careful attention and a certain level of expertise, especially when searching for the best-performing algorithm, selecting the most appropriate data structure, or struggling with memory hierarchy. Yet, this process is necessary to identify opportunities for improvement that may result in minor reductions in execution time. LLMs can be used as a way to make this process easier, *e.g.*, by generating performance-improving code edits [11, 15, 25]. Garg *et al.* [16] also presented RAPGen, a method for generating zero-shot prompts to enhance performance. However, while users seem to put a lot of trust in code generated by LLMs, they still have trouble reviewing it [33, 34], which can lead to slow code being shipped in production, especially if an LLM generates inefficient code.

The key contributions of this paper are as follow : (i) We study the performance of the code generated by 18 LLMs on 204 problems and investigate performance differences across models using a novel method for measuring and comparing the performance of LLM-generated code. (ii) We compare the performance of the code generated by LLMs to the code written by humans. (iii) Incidentally, we evaluate the usability of Leetcode,[4] a public repository of algorithmic problems that we use as a dataset.

From section 2 to section 4, we describe the tasks dataset and the models we selected, outline our experiment setup, and explain the methodology we followed to analyze the obtained results, respectively. We report in section 5 on the results of our evaluation and provide a critical discussion in section 6. Finally, section 7 presents the related works, and section 8 concludes the paper.

[4]https://leetcode.com/

2 METHODOLOGY

2.1 Research questions

This paper covers the performance of code generated by various LLMs. In particular, we aim to answer the following research questions:

RQ1: *Can Leetcode be used as a dataset and a benchmark platform for evaluating LLMs?* Leetcode can serve as both a dataset of problems and a tool to evaluate and measure solutions to the problems. Particularly, we study if the dataset is subject to recitation and if the measures Leetcode provides are reliable.

RQ2: *Are there notable differences between the performance of the code generated by different LLMs?* LLMs differ greatly in terms of generating correct code, so we want to know if they also differ in terms of generating efficient code.

RQ3: *Is there an effect of the success rate and the temperature of the LLM on the code's performance?* Having a higher temperature decreases the capacity of the LLMs to generate valid code, so we aim to study if this also applies to the performance of the code. In the same way, we also study if an LLM that is very good at generating valid code on one problem, is going to generate efficient solutions to this problem.

RQ4: *How efficient are the solutions generated by the LLMs compared to human solutions?* Comparing the LLMs to a set of human-authored solutions can provide insights into their position relative to humans in terms of code performance.

2.2 Tasks & Dataset

Task selection. The input tasks—*i.e.*, problems specified by a prompt—we consider to generate code from various LLMs has to meet the following requirements:

- A given problem, should offer multiple candidate solutions, whose generated code performs differently. This ensures one can observe differences across the various LLMs;
- Generated solutions should exhibit variable execution times. Given more complex inputs, one should differentiate $O(n)$ from $O(2n)$ and $O(n^2)$ algorithms.

As a result, task datasets, such as HumanEval or *Mostly Basic Python Programming* (MBPP), which are classically used when evaluating LLMs for code assessments [1, 4, 10, 27], cannot be considered for our purpose. Indeed, while they do provide unit tests to drive the generations, the size of the inputs in these unit tests remains small and fails to scale to appreciate performance issues. Moreover, the solutions that need to be generated are often very short, which would lead to fewer possible variations between implementations. Also, the fact that many problems are not algorithmic by nature makes them less prone to inefficient practices and performance variation. To face such issues, we used input prompts that were built from Leetcode problems. Leetcode is an online judge platform that suggests programming problems to registered users. It addresses the above limitations as it provides algorithmic problems with varying levels of difficulty and test cases with large input sizes. Leetcode also exposes a GraphQL API[5] to fetch relevant metadata on the problems, such as exercise instructions, code snippets containing the signature of the function to generate, as well as the difficulty and topics of the problem.

We followed an experimental design similar to the one used by Döderlein *et al.* [13], while using a different set of Leetcode questions. To avoid data contamination, which happens when an LLM is tested on data it was trained on,[6] we only considered problems that were published after January 1^{st}, 2023. As all the LLMs (except GitHub Copilot) we evaluated were trained using datasets older than these problems, we avoid any data contamination. As these problems are published by Leetcode in the context of programming competitions, they are always original. However, GitHub Copilot being an online closed-source tool, one cannot tell whether it underwent training with the problem set we employed. Our set was composed of 204 problems—labeled as 56 easy problems, 104 medium problems, and 44 hard problems, as classified by Leetcode upon the problem's publication.

To answer RQ1, we also performed our experiment a second time on the set of questions used by Döderlein *et al.* [13], which is composed of 300 problems (95 easy, 105 medium, and 100 hard) from the most liked problems of Leetcode. We will refer to this dataset as the "old" dataset, and to our dataset of problems published during 2023 as the "new" dataset. Code generation was performed in Python for its ease of use and because of the prevalence of Python-written datasets for evaluating LLMs [5, 10].

Input prompts. The instructions given by Leetcode for each programming problem contain *(i)* the description of the problem,

[5]https://pypi.org/project/python-leetcode/

[6]You can find more details on data contamination here.

```python
# Start of the input prompt
"""
Given an integer array nums, return all the triplets `[nums[i],
↪  nums[j],nums[k]]` such that `i != j`, `i != k`, and `j != k`,
↪  and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.
"""
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # End of the input prompt

        # Start of the generated code
        nums.sort()
        res = []
        for i in range(len(nums) - 2):
            ...
        # End of the generated code

# Start of the benchmarking code
def check():
    Solution().threeSum([82597, -9243, 83030, ...])
    Solution().threeSum([0, 0, 0, 0, ...])
    Solution().threeSum([0, 0, -1, -1, ...])

import pytest
@pytest.mark.benchmark(group="3sum")
def test_3sum_generated_1(benchmark):
    benchmark(check)
# End of the benchmarking code
```

Figure 1: Example of problem's input prompt, generated code, and benchmarking code.

(ii) examples of inputs and outputs, and *(iii)* constraints on the input data. To build the prompts inputted to the LLMs, we chose to only include the description of the problem. This choice aimed to maintain the prompt's conciseness and ensure compliance with potential LLMs' length restrictions. Indeed, LLMs are constrained by a context size, imposing the maximum number of tokens they can process at once, encompassing both the prompt and the answer. In our study, the majority of LLMs impose a context window of 1024 tokens. Figure 1 illustrates a descriptive prompt (enclosed within triple quotes) along with a solution generated using this prompt and the corresponding benchmark instructions. Leetcode occasionally includes an additional comment in the prompt, indicating the available methods of the elements passed to the solution (*e.g.*, binary trees). While adding examples in the prompt could potentially increase the likelihood of generating correct solutions [13], we chose not to include them due to the complexity of representing Leetcode's data structures (such as arrays, graphs, linked lists, trees, etc.) in textual form. On Leetcode's website, data structures are textually represented using an array notation format with brackets (*e.g.*, "`[1, 2, 3]`"), which might be misleading for an LLM working with data structures that differ from arrays or lists.

Canonical solutions. Each problem was matched with a single valid solution, written by a human and fetched from various sources. These solutions, referred to as "canonical solutions", are considered as baselines during the benchmarking process, although they may not represent the entirety of human-written solutions. They were used to assess the stability of the measuring process. Most of the canonical solutions were fetched from the WalkCC repository of Leetcode solutions.[7] When one question did not come up with any solution in Python from this repository, we selected one solution proposed by the Leetcode community among the most upvoted ones (starting from the one with the most upvotes), which are also publicly available on the Leetcode website. These modifications only included changes to the function name, variable names, and type hints except for one specific case, where a recursive solution provided by WalkCC was replaced by an iterative one from the Leetcode community because of the stack limit on our local setup. We ended up with one Python-based canonical solution for each problem of our dataset.

Test cases. Testing generated solutions is a twofold process, as both the correctness and scalability (performance) of each solution must be checked. First, to ensure that generated solutions are correct, we provide such solutions to the Leetcode online judge system, which in turn validates them based on its test suites. Second, we execute the valid solutions with input data to assess their performance. To retrieve such input data, we crawled through the problems' instructions and extracted two to three examples of inputs and expected outputs provided by Leetcode. However, such inputs were too small to exhibit significant performance differences, *e.g.*, arrays containing only two to five elements. To execute the generated solutions with larger input data, we took advantage of Leetcode's judge system that returns the inputs and expected output of the first failed test of a test suite. We noticed that Leetcode tests if the submitted solution is inefficient by using a timeout system. Since all selected problems are algorithmic by nature, one simple way to put a heavier load on their implementations is to increase the size of the inputs. For every problem, we thus submitted a modified version of a canonical solution that failed only when the size of the first parameter exceeded a certain threshold. We then set this threshold manually multiple times for every problem to extract three different inputs for each problem, resulting in more than 150 MB of fetched input data with most inputs having over 10^5 elements.

[7] https://github.com/walkccc/LeetCode

2.3 LLMs Under Study

Our empirical study covers a total of 18 LLMs, specifically designed for coding purposes. We selected the 18 popular code LLMs from Hugging Face,[8] as well as GitHub Copilot, which is an online closed-source code assistant. The LLMs were selected based on the number of downloads and likes they exhibited. We chose GitHub Copilot to offer a comparison between a commercial LLM and open-source LLMs, but did not choose any other GPT models from OpenAI because of their cost. Table 1 summarizes all the LLMs considered in this study. This includes variants of the same base LLM—*i.e.*, models with varying sizes (in billions of parameters) or different training data, such as variants of CodeGen, InCoder, and CodeT5. LLMs belonging to the same family are models closely related in terms of training data and method. We first performed our experiment in March 2023 with a subset of the models presented here, with the "old" dataset. We then performed it a second time in September 2023 with all the models and the "new" dataset.

LLM Model	Model family	Size	RQ1
GitHub Copilot	Codex	11	✓
CodeGen-Mono 6B	CodeGen	6	✓
CodeGen-Mono 2B	CodeGen	2	✓
CodeGen-Mono 350M	CodeGen	0.35	✓
CodeGen2.5-7B-mono	CodeGen2.5	7	
CodeGen2.5-7B-instruct	CodeGen2.5	7	
CodeLlama-7B-instruct	CodeLlama	7	
CodeLlama-7B	CodeLlama	7	
CodeLlama-7B-python	CodeLlama	7	
CodeLlama-13B-instruct	CodeLlama	13	
CodeLlama-13B-python	CodeLlama	13	
replit-code-v1-3b	replit-code	3	
WizardCoder-pythin	WizardCoder	7	
SantaCoder	Santacoder	1.1	✓
StarCoder	StarCoder	15.5	
InCoder 6B	Incoder	6	✓
InCoder 1B	Incoder	1	✓
CodeParrot	Codeparrot	1.5	✓

Table 1: LLMs considered in our study. Models with the RQ1 checkmark were also evaluated on the "old" dataset. Size is in billions of parameters

3 EXPERIMENT SETUP

This section describes our experiment setup to generate and validate the solutions produced by the LLMs. First, we describe how solutions were generated from each LLM. Then, we outline the three-step process used to filter invalid solutions. Finally, we explain how the run time of the generated solutions was measured.

3.1 Code Generation

We generated 10 solutions for each problem from our dataset by varying the temperature of the LLMs used to generate them. Specifically, we considered 6 different temperatures (`0.1`, `0.2`, `0.4`, `0.6`,

[8] https://huggingface.co

`0.8`, and `1.0`). As Copilot's temperature cannot be configured, we used its default temperature for generating all the problems.

Generating code with GitHub Copilot. Automatically generating with GitHub Copilot for a reproducible experiment proved to be a difficult task. Firstly, the code suggestion feature of Copilot activates when typing in a text editor with the installed Copilot plugin. Additionally, GitHub Copilot produces code based on a context that encompasses the current file and the files previously accessed by the user, impacting the generated solutions. Lastly, we noticed a caching mechanism on the GitHub Copilot server side, which resulted in very similar or identical solutions if we generated multiple solutions for a given problem in the same session. To address these issues, we used a generation method similar to the one used by Döderlein *et al.* [13] by instrumenting the GitHub Copilot Neovim plugin[9] and restarting the plugin between every generation to avoid the caching effect. This method allowed us to automatically generate solutions in a quick and isolated fashion. On top of that, GitHub Copilot provides 2 means of generation: *inline* generations (the suggested code is integrated with the editor) and *panel* generations (Copilot generates at most 10 completions and displays them on a panel next to the editor). We chose to exclusively use inline generations, as panel generations yielded worse results in terms of functional correctness than inline generations.

Generating code with open-source models. Regarding the open-source models, we generated solutions by deploying the models on servers provided by the Grid5000 platform [6]. We used the Deepspeed library to make the generation process faster and fit larger models on our GPUs. Concerning the sampling, we used the same methods as Chen *et al.* [10] and used nucleus-sampling [18] with top $p = 0.95$. The maximum number of tokens to be generated was set to 600. This is because the LLMs we used have a limited context size of 1024 tokens (prompt included). Thus, to avoid exceeding the context size, we had to limit the number of tokens to generate. We also verified it did not significantly impact the functional validity of the LLMs.

In total, we generated 2,040 solutions with Copilot and 12,240 with each of the eight other models, resulting in 210,120 generated solutions overall.

3.2 Validation

Each generated solution was tested to ensure its functional correctness following a three-step process. At every step, if a solution was found to be invalid, it was excluded from subsequent stages of the experiment. The process was as follows:

i) Local validation. We filtered out code generations that included easy-to-spot errors, such as syntax errors or runtime errors, by using the small inputs we fetched earlier (see Section 2.2 - *Test cases*). While this step was not strictly necessary, it quickly reduced the number of solutions to be validated in the next step;

ii) Leetcode validation. Next, we submitted the solutions to the Leetcode judge system using the Leetcode GraphQL API, where they underwent a rigorous test suite managed by Leetcode. We kept the solutions that passed all the test cases or exceeded Leetcode's allocated time limit. The latter was kept to ensure that correct solutions that were too slow remained included in the benchmark;

iii) Exclusion of timeouts and other errors. Finally, we excluded the solutions that reported errors when executed with our benchmarking setup using the large inputs fetched beforehand. We invalidated the code generations that took more than 10 seconds to run or raised an error. Most of the errors raised in this step were recursion errors caused by differences between Leetcode's Python interpreter and ours. Indeed, Leetcode's seemed to have a higher recursion limit than ours, which we set to 10,000 instead of the default 1,000 (we could not manage to set it any higher). Additional errors occurred because we developed our helper classes differently from Leetcode's implementation. Although these classes are provided by Leetcode during the submission process, they are not publicly available. Out of the 4,930 invalidated solutions that were caught in this step, 4,863 (98.6%) were due to timeouts, 20 (0.04%) to recursion errors, and 47 (0.1%) to other errors. Following this validation process, the initial set of 210,120 generated solutions was pruned down to 7,481 (3.6%) valid solutions remaining across the 18 LLMs.

3.3 Measuring run time

We measured the performance as the run time of the generated solutions using pytest-benchmark,[10] which runs pytest unit tests multiple times to obtain run times statistics. The measurements were performed using parameters that ensured each solution ran at least 10 times and for at least 1 second in total. We did not perform warm-up runs of the benchmarks, as we did not notice any significant difference in the measured time during preliminary testing. To facilitate the measurement protocol, the generated solutions were sorted into "runs", based on the specific LLM and temperature that were used during the code generation. Within each run, which was defined by a unique combination of model and temperature, the solutions were measured in sequence during a single program execution. Furthermore, in every run, we added the canonical solutions we previously collected, which would run alongside the generated solutions. This approach ensured that the same canonical solutions were executed in every run, allowing us to maintain measurement stability. Specifically, we calculated the standard deviation of the canonical solution run times across all runs, thus providing a reliable measure of the variability of the measurement protocol. We observed that over 96% (196 out of 204) of the canonical solutions had a standard deviation lower than $1/10^{th}$ of their average run time, which we deemed to be an acceptable level of variation.

The cluster we used to run the benchmark was the chiclet cluster of the Grid5000 testbed.[11] It hosts 2 AMD EPYC 7301, with 16 cores per CPU and 128GB of memory. When using the node, all the cores of both CPUs were reserved, but only one was used at a time to maximize the stability of the measurement protocol.

3.4 Replication package

All the artifacts of this study, including our results, code, and datasets, are available in the following public repository: https://zenodo.org/doi/10.5281/zenodo.7898304.

[9] https://github.com/github/copilot.vim

[10] https://github.com/ionelmc/pytest-benchmark

[11] http://grid5000.fr

4 DATA ANALYSIS

In this section, we describe the methods we adopted to analyze our results. These methods fall into one of the two following categories: functional correctness and code performance.

4.1 Functional Correctness

The functional correctness of an LLM defines how much the LLM outputs code conforming to the program contract (as specified by the input prompt). To evaluate the functional correctness of our LLMs, we computed their pass@k metrics with $k = 1$ and $k = 10$, using the unbiased estimator proposed by Chen *et al.* [10]. The pass@k unbiased estimator which, from k samples produced, considers the test as successful if one of these samples passes all the tests, is computed as follows (with n the total number of samples, c the number of correct samples and $\mathbb{E}$ the expected value):

$$pass@k := \mathop{\mathbb{E}}_{Problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (1)$$

As Chen *et al.* [10] suggest, we calculated the pass@k for each temperature when evaluating an LLM's functional correctness and considered the best one as the pass@k for that LLM.

4.2 Code Performance

To measure the code performance, we considered three different metrics. First, we used the memory usage reported by Leetcode. Then, we computed the median of the run times measured by pytest-benchmark for every generated solution. Lastly, to compare the LLMs solutions to human-submitted solutions, we also used the rank reported by Leetcode when validating the solution. This rank is a number between 0 and 100 that indicates the share of submitted solutions that are slower than the current solution (*e.g.*, if a solution has a rank of 90, it is faster than 90% of the submitted solutions on Leetcode).

To assess the LLMs' performances, we conducted pairwise comparisons as follows: For each pair of LLMs, we identified problems where both models generated more than 5 valid solutions. For each identified problem, we conducted a Student t-test on the mean run time of the generations to determine if there was a significant difference. Then, for each pair A-B of LLMs, we computed the ratio of problems where A's code was significantly faster than B's and where B's code was significantly faster than A's.

5 RESULTS

In this section, we summarize the key observations from our experiment and answer our research questions. In Table 2, you can also find the functional validity results of the different LLMs on both of our Leetcode datasets. Our results are also available in the form of a companion notebook in our replication package. The companion notebook offers more insight into the results and additional graphs.

5.1 RQ1: Can Leetcode be used as a dataset and a benchmark platform for evaluating LLMs?

5.1.1 Can Leetcode problems be adopted as a dataset for LLM generation? As on can observe in Figure 2, the generated codes exhibit on a significant drop in functional correctness between the two datasets. The difference here is pretty staggering for every tested LLM, reporting on a tenfold decrease in pass@k. We believe this issue may stem from data contamination in the old dataset. Data contamination occurs when an LLM is assessed on data that was included in the training dataset, introducing bias into the evaluation process. In our case, a significant number of questions in the old dataset are widely known and have been extensively shared on GitHub. For instance, a search for the prompt of the "3sum" Leetcode problem on GitHub yields approximately 4.000 matches in public repositories. These questions are also old enough to likely be included in the training datasets of the LLMs under study, as the majority of their training datasets have a cut-off date between 2021 and 2022. Due to this data contamination, LLMs tend to recite, reproducing verbatim source code when generating solutions. This phenomenon is more pronounced when the prompt is highly specific and lacks contextual information, as seen in Leetcode prompts

LLM Model	Pass@1	Pass@10
StarCoder	0.095	0.132
CodeLlama-13B-python	0.093	0.201
GitHub Copilot	0.092	0.196
CodeLlama-7B-instruct	0.082	0.191
CodeLlama-13B-instruct	0.078	0.206
WizardCoder-python-7B	0.075	0.157
CodeGen2.5-7B-mono	0.066	0.147
CodeGen2.5-7B-instruct	0.062	0.142
CodeLlama-7B-python	0.047	0.172
CodeGen-6B-mono	0.045	0.113
CodeGen-2B-mono	0.038	0.103
replit-code-v1-3b	0.025	0.083
InCoder-6B	0.021	0.064
SantaCoder	0.015	0.064
CodeLlama-7B	0.014	0.015
InCoder-1B	0.012	0.039
CodeGen-350M-mono	0.007	0.039
CodeParrot	0.002	0.015

Table 2: Functional validity of the LLMs on Leetcode (by decreasing pass@1**, higher is better)**

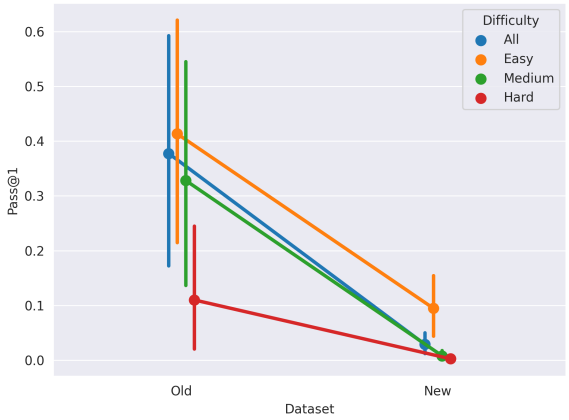

Figure 2: Average pass@1 **of the evaluated LLMs for every difficulty and dataset, with 95% confidence interval (higher is better)**

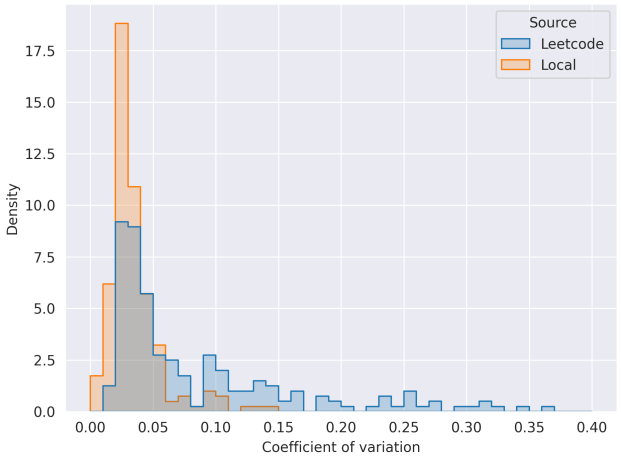

Figure 3: Coefficicent of variation of the time measured by Leetcode and locally for every problem using canonical solutions

Figure 4: Scatter plot of the measures done by Leetcode and locally for every generation for the problem "*Difference between element sum and digit sum of an array*". Orange points are from multiple measures of the same canonical solution and serve as visual references for the measurement error

that closely match GitHub repositories. The observed shift in functional validity between the two datasets could also arise from a genuine difference in the difficulty of the questions within each dataset. However, quantifying this last hypothesis proves to be challenging.

5.1.2 Are Leetcode measurements reliable? **Run time.** As reported in Figure 3, the coefficient of variation of the Leetcode measures (0.089) is slightly higher than the coefficient of variation of the local measures (0.035). This suggests that Leetcode's measuring setup is less suited to ensure accurate benchmarks.

We also study the correlation between our local measurements and Leetcode's, and we notice two issues: (1) the times measured locally and by Leetcode only slightly correlate on average (0.28). For some problems, the measures are highly correlated (> 0.8) while, for others, they are almost not (< 0.2). This is more apparent when we look at the scatter plot showing the measures of some problems in Figure 4. In this problem, there are four clusters of generations with a different locally measured time, but the clusters are indiscernible in terms of Leetcode time. This could be due to two main reasons. Firstly, as previously discussed, the variance of the measures from Leetcode is higher and as such, there is much more noise in the measures, decreasing the precision. Secondly, the tests we employed may be more focused on performance testing than Leetcode's. Notably, our test suite comprises only three tests featuring significantly large inputs, potentially accounting for certain disparities in the results.

Although still usable, relying on the time reported by Leetcode introduces some limitations due to its higher variance. Consequently, Leetcode's measures cannot allow us to discern the differences in run time between different code implementations as precisely as locally measured time.

Memory usage. While we did not measure the memory ourselves, we observed the variation of the memory usage measure provided by Leetcode. We notice that the memory usage for the same solutions decreases over time. There is indeed a slight correlation of -0.24 between the day of the year we tested our solution and the memory usage. The fact that the memory usage measure evolves renders comparisons between LLMs harder. While we could theoretically offset the memory usage when we detect changes over time, it would require testing the canonical solutions alongside the generated one on Leetcode.

Leetcode rank. The time ranking that Leetcode returns when we test a solution represents the share of submitted and valid solutions that are slower than ours. While this could be a great tool to rank LLMs among human-submitted solutions, we find that the ranking is heavily affected by our submissions and time. As you can see in Figure 5, the overall rank of the LLMs we tested decreases over time. To verify this, we tested GitHub Copilot twice, once as the first LLM, and a second time after testing all the LLMs. The first test of Copilot has an overall rank of 77, and the second test ranks down to 54, despite it being tested with the same solutions. This effect of the rank evolving becomes obvious when you consider that the rank is determined using all the previously accepted solutions, including ours. This means that by testing thousands of our solutions on Leetcode, we are actively changing the ranks of future tests.

> **RQ1**: The evaluation of LLMs using Leetcode questions as a dataset presents some challenges. Although Leetcode's questions could serve as a valuable dataset akin to HumanEval, limitations arise due to the constraint that only the problems published after the LLM's training dataset formation are usable for evaluating the LLM in question. This creates potential difficulties in reproducibility, particularly as new LLMs emerge, especially if they do not exclude Leetcode problems from their training datasets [19]. Additionally, while Leetcode's provided metrics, such as run time, memory usage,

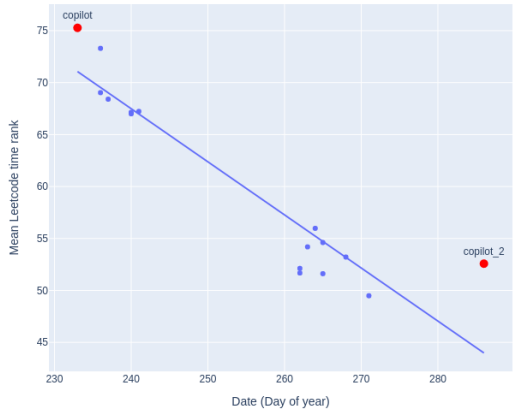

Figure 5: Scatter plot of LLM's rank and date they were tested on Leetcode. The two models in red are the same model tested on different dates

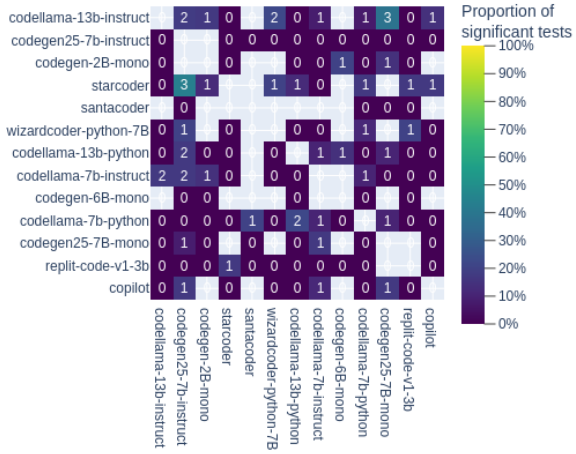

Figure 6: Number of problems where an LLM (row) is better than another (column)

> and rank may offer practicality in various scenarios, their usability and reliability are questioned when compared to more traditional measurement methods. The presence of these challenges emphasizes the need for careful consideration and scrutiny when adopting Leetcode to evaluate LLMs.

5.2 RQ2: Are there notable differences in performances between LLMs?

The pairwise comparison depicted in Figure 6 reveals subtle distinctions in the performances of various LLMs. Notably, some models, such as StarCoder and the CodeLlama model with 13B parameters specialized in Python, consistently exhibit slightly superior results compared to others. Despite these observed variations, the mean Cohen's d effect size measures a mere 0.024, a statistically insignificant magnitude. This suggests that the practical impact of these differences on the mean speed of code generation is remarkably small. For instance, when comparing CodeLLama-13-instruct and CodeGen25-7B-mono, CodeLLama outperforms the latter in a statistically significant manner in 3 problems out of 8. However, it is crucial to note that the mean performance difference between these models is a mere 0.02 standard deviation. It thus seems that improving an LLM in terms of functional validity does not significantly impact the performance of the code it generates. This may be due to different factors, such as the fact that most LLMs share the same datasets or that they are trained to produce valid code and not fast code. Improving the performance of an LLM could be done by curating a training dataset of only efficient code and fine-tuning one of the foundational models we used, or by using reinforcement learning to "teach" the model to produce better code. Madaan *et al.* produced an LLM that could improve the performance of code [25]. This LLM could be leveraged in a generation pipeline to directly improve the generated code.

> **RQ2**: Our analysis uncovers statistical differences in the performance of generated code among different LLMs. However, the effect size, as measured by Cohen's d, is so negligible that it raises questions about the practical significance of these differences. Despite some models consistently outperforming others, the overall impact on the mean efficiency of LLM-generated code appears to be minimal.

5.3 RQ3: Is there an effect of the functional validity of the LLM and its temperature on the generated code's performance?

Functional validity. When calculating for every problem the correlation between the success rate of the LLM that generated the solution and the run time of the solution, we find that there is only a very slight negative correlation (−0.08) between the success rate and the performance. There is close to no correlation (−0.11) observed between the success rate of the model and the variation in the performance of the generated code.

Temperature. There is no correlation (0.05) observed between the temperature of the generations and the performance of the generated code. This means that the temperature does not affect how fast the solutions are. However, we observe that temperature is moderately correlated (0.41) with higher variations in performances. This means that higher temperatures tend to increase the variation in performance across generations. The complete distribution of the correlation for each of the 24 problems can be seen in Figure 7. So, while increasing the temperature leads to a lower success rate [13], it can help find a faster solution with an extended exploration of generations. The fact that the temperature increases the variation in performances comforts the idea that higher temperatures lead to more diverse outcomes.

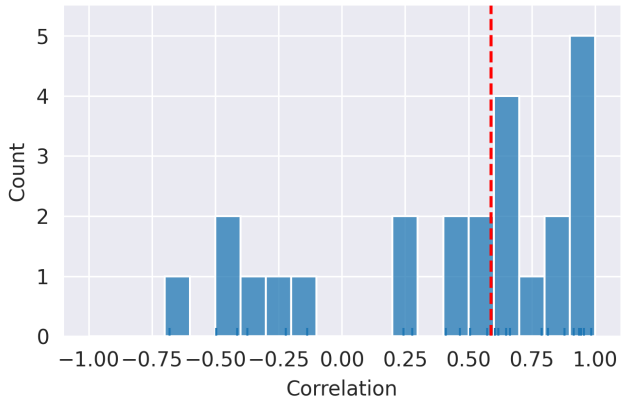

Figure 7: Distribution of correlations for every problem between the temperature and the variation of the performance. The red line is the median

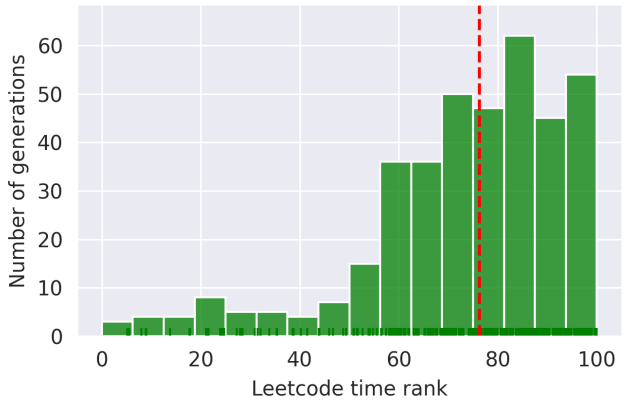

Figure 8: Distribution of the ranking for the CodeGen-6B-mono model

> **RQ3**: Our analysis of LLMs indicates that the quality of generated code does not have a substantial impact on its performance. However, we observed that modifying temperature settings within an LLM significantly affects the diversity of code performances produced. This implies that, while code validity may not be a decisive factor in performance, adjusting temperature settings can be a valuable strategy to enhance the variety of outcomes in code generation processes.

5.4 RQ4: How fast are LLMs compared to humans ?

As previously stated, the Leetcode time ranking evolves, so we chose to compare the second model we tested on Leetcode with humans (Copilot being the first, it did not have enough generations overall because of its lack of a temperature setting). The results of this comparison are depicted in Figure 8. The comparison is done using the Leetcode ranking, with the assumption that most of the previous submissions were made by humans.

We observe in Figure 8 that the solutions generated from LLMs are faster than most previous submissions with a mean rank of 73%, and that it even generated some solutions that were faster than 95% of the previous submissions.

> **RQ4**: It seems that the LLMs are faster than most of the human solutions on Leetcode, on average. If the LLM we tested were in an actual competition, his valid solutions would be on average faster than 73% of the other solutions on Leetcode.

6 DISCUSSION

In this section, we discuss the results reported in the previous section, their implications, and the limitations of our study.

6.1 Discussion of the results

On the Leetcode measures and usability. The data contamination issue we unveiled poses a significant challenge in the evaluation of LLMs, as it prevents an accurate evaluation of their real performances. Because Leetcode's problems are not filtered from the training datasets, as research on LLMs continues, even the newer problems might contaminate future training datasets, thus rendering reproduction of our study harder [19]. This conclusion also holds for any study using Leetcode as an evaluation dataset, such as [8, 13, 26]. However, we believe that the methodologies employed and the conclusions drawn would likely hold validity with alternative sets of questions from Leetcode or other performance-oriented datasets. This suggests that future studies seeking to replicate our findings would primarily need to change the dataset employed for assessing LLMs. Addressing the data contamination concern could involve leveraging pre-filtered evaluation datasets, such as HumanEval, already separated from the training processes, and repurposing them into performance evaluation datasets.

On functional correctness. The ranking of the functional correctness of the LLMs is consistent with the previous evaluations of the LLMs on HumanEval [1, 4, 10, 21, 24, 27, 32]. However, it seems that InCoder performs worse than presented in its introductory paper [14], which may be due to our experimental protocol—using it only for left-to-right generation instead of infilling like it was built for.

We observe that Leetcode's problems seem harder for the LLMs to solve than HumanEval's problems. Indeed, StarCoder, the model that performed the best with a pass@1 of 0.09, had a pass@1 of 0.408 on HumanEval [21]. We believe this is due to multiple factors. First, our Leetcode prompts and expected solutions are longer than in HumanEval (the average length of solution in HumanEval is 180 characters *vs* 425 characters in our dataset), thus increasing the chance of a generation to fail. Indeed, having to generate more code can lead to a higher chance of making a mistake, as shown by a correlation of −0.30 between the solution's length and the success rate of the problem. Second, Leetcode problems come from programming competitions and need a lot of thinking to be solved. The

causal generation of the LLMs does not allow a "thinking" process to happen, which for harder problems causes a drop in functional validity. This could be solved by making the LLMs mimic human thinking with methods, such as Chain-of-Thought prompting [39].

On the performance of generated code. To the best of our knowledge, our methodology for evaluating LLMs based on the performance of generated code is novel and could serve as a benchmarking approach for future studies involving new LLMs and datasets. While we aimed for a singular performance score for LLMs, similar to pass@k, the varying rates at which LLMs generate valid solutions posed a challenge, leading us to employ pairwise comparisons. An improvement to our method could involve using an optimal solution as a reference point and comparing the speed of each generated solution to the optimal solution's speed. Speed would thus be expressed as a factor relative to the optimal time, addressing the issue of problems having different time scales and potentially laying the groundwork for a more comprehensive "performance score."

The low success rate of LLMs on Leetcode posed a considerable challenge for their comparison. Among the 204 problems, only 24 had *(i)* valid solutions from at least 10 different models (out of the 18 assessed LLMs) and *(ii)* at least 10 valid solutions in total (see to the companion notebook). This low success rate complicated pairwise comparisons, as numerous models could not be compared due to an insufficient number of common problems with a substantial number of generated solutions.

Our discoveries offer valuable insights for developers in their choice of LLMs. For example, when developers are considering an LLM for tasks like code generation, such as with GitHub Copilot, the performance of the generated code may be an important concern. With our findings, developers can be assured that there is no significant variance in the performance of code generated by different LLMs. This means that if they aim to have fast code, there's no necessity to consistently opt for the largest model available; instead, a smaller one suffices. We also hope that our findings will incentivize further research on building new LLMs that produce even more efficient code.

6.2 Limits and Threats to Validity

Regarding functional correctness, although we did our best to generate code in the most optimal conditions, some changes to the input prompt (*i.e.*, adding examples and constraints), or the configuration of the models may have changed the performance of the LLMs. However, we believe that this should not significantly impact the validity of our experiment, as all the models were configured similarly.

GitHub Copilot being a closed-source tool, it might be retrained without the community's knowledge, which could potentially lead to a modification in the tool's performance in upcoming experiments.

The majority of our validation and benchmarking process relied on Leetcode's test suite and online judge system. Thus, it is possible that some big test cases we extracted for the benchmarking favored some types of implementations over others. We mitigated this issue by systematically fetching three test cases and by having a large dataset of problems to generate from. During our experiments, we also tried to generate plausible inputs using random generators, but this method did not yield satisfying results because randomly generating data structures with specific shapes, or properties (*e.g.*, generating valid regular expressions) is difficult to achieve.

One gap in our study is that we do not consider memory usage at all when studying the LLMs. This is because our benchmarking setup did not allow for memory monitoring and doing otherwise would have cost us a lot of time. Moreover, we believe our results obtained with only the run time to be self-sufficient.

The way we compared an LLM to a human using Leetcode rankings gives a good idea of where the LLM stands in terms of performance. However, because the ranking evolves and we have no information about the population the LLM is ranked against, the results here should be taken with a grain of salt.

It is also important to note that Leetcode as an evaluation dataset suffers from some issues. As we only evaluate the LLMs on algorithmic problems, the performances of the LLMs are hard to generalize across all programming fields. However, this is difficult to improve on for similar reasons to HumanEval: we only have a limited context size and have to make the LLM generate in one go a completion to some code, which must be self-contained (meaning, all the information needed to generate the solution must be in the prompt). Also, in terms of performance, it is difficult to find another kind of self-contained code than algorithmic problems that have, such variations in performance. While studying the LLMs on SQL generation could be a good idea, it would not fit into our studies with generalist programming languages.

Regarding Leetcode as a platform, we are heavily dependent on it for validating the generated solutions and are limited by its daily rate limits of $1,000$ submissions per account. For future studies, it would be great to consider alternatives to Leetcode, such as the project CodeNet [31], which does not have as many restrictions as Leetcode.

7 RELATED WORKS

Previous research has investigated various aspects of LLMs for code-related tasks, including the security of their suggestions [29, 30, 33], the prevalence of bugs in the generated code [20], how developers interact with them [7, 30, 34] or just the quality and correctness of the code they generate [13, 22, 26, 42]. There have also been efforts to measure the efficiency of LLMs through the creation of benchmarks for comparing them, such as HumanEval[10], MBPP [5], CoderEval [43], APPS [17], CodeXGLUE [23] or ReCode [37]. Xu *et al.* [40] also led a comparative evaluation of multiple LLMs for code including Codex and Codeparrot.

Leetcode, while being just a coding competition platform, is also used as a dataset to evaluate the capabilities of LLMs on programming tasks. Döderlein *et al.* [13] measured the performances of Copilot and Codex on Leetcode and the effects of changing the prompts. Nguyen and Nadi [26] studied Github Copilot's code suggestions on Leetcode problems and the complexities of its generated code. Vasconcelos *et al.* [35] studied the effects of highlighting the uncertainty of AI-powered code completions using Leetcode problems and Codex.

Various other methods have also been employed to investigate the impact of temperature on the generated code, apart from the

approach we proposed: Chen *et al.* [10] evaluated the best temperature of Codex in terms of pass@k. Austin *et al.* [5] also studied the effects of the temperature on the performance of their model. Christopoulou *et al.* [12] also studied the effects of the temperature and nucleus-sampling on their LLM. Döderlein *et al.* [13] highlight the importance of correctly tuning the temperature of a model when using it to generate code. The research led by Aghakhani *et al.* [3] shows that poisoned models suggest insecure code more often as the temperature increases. Our results demonstrate that increasing the temperature also increases the chance of generating slow and inefficient code.

On the subject of performance, Madaan *et al.* [25] fine-tuned LLMs to make them improve the performance of code. Multiple other techniques have been proposed for automatically improving the performance of code using LLMs [9, 11, 15]. Our contribution is the first—as far as we know—to investigate the differences in the performance of LLM-generated code. Future LLMs that would be adapted with these performance-improving techniques could also be compared using our methodology. Regarding recitation, few works have been done on this subject, but Yan *et al.* [41] proposed a method to detect cases of recitations in LLMs using inference fingerprinting. Jacovi *et al.* [19] explained why data contamination in LLMs was problematic and proposed methods to mitigate it.

To summarize, our paper evaluates the performance of code generated by various LLMs and investigates differences in the performance of the generated code across models on Leetcode problems, which, to the best of our knowledge, has not been done previously.

8 CONCLUSION

In this study, we presented a comprehensive analysis of the performance of code generated by various LLMs using a novel methodology that measures and compares the runtime speed of solutions to algorithmic problems. Our findings suggest that the performance of the generated code is largely similar across different models, regardless of their size or training data. Furthermore, increasing the temperature parameter during code generation leads to a greater variance in performance, though not necessarily to better or worse solutions on average. We also critically evaluated the suitability of Leetcode as a dataset and benchmark platform for assessing LLMs. The results indicate that, while Leetcode's problems are suitable for performance evaluation, their measures should be used cautiously due to issues with stability and reliability. Additionally, we observed that the use of newer Leetcode problems is essential to avoid data contamination and ensure the validity of LLM evaluations.

This work opens up several avenues for future research, including the development of performance-oriented training datasets and the fine-tuning of LLMs for performance improvement. As the field of AI-assisted programming continues to evolve, studies such as ours will play a critical role in understanding and enhancing the capabilities of LLMs.

ACKNOWLEDGMENT

This work received support from the French government through the *Agence Nationale de la Recherche* (ANR) under the France 2030 program, including partial funding from the CARECloud (ANR-23-PECL-0003), DISTILLER (ANR-21-CE25-0022), and KOALA (ANR-19-CE25-0003-01) projects. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.[12]

REFERENCES

[1] 2022. Codeparrot/Codeparrot · Hugging Face. https://huggingface.co/codeparrot/codeparrot.
[2] Hayri Acar, Gülfem I Alptekin, Jean-Patrick Gelas, and Parisa Ghodous. 2016. The Impact of Source Code in Software on Power Consumption. *International Journal of Electronic Business Management* 14 (2016), 42–52.
[3] Hojjat Aghakhani et al. 2023. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models. (2023). https://doi.org/10.48550/ARXIV.2301.02344
[4] Loubna Ben Allal et al. 2023. SantaCoder: Don't Reach for the Stars! arXiv:2301.03988 [cs]
[5] Jacob Austin et al. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs]
[6] Daniel Balouek et al. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*. Communications in Computer and Information Science, Vol. 367. 3–20.
[7] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 78:85–78:111.
[8] Sébastien Bubeck et al. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. arXiv:2303.12712 [cs]
[9] Binghong Chen and othersy. 2022. Learning to Improve Code Efficiency. arXiv:2208.05297 [cs]
[10] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs]
[11] Zimin Chen, Sen Fang, and Martin Monperrus. 2023. Supersonic: Learning to Generate Source Code Optimizations in C/C++. arXiv:2309.14846 [cs]
[12] Fenia Christopoulou et al. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. arXiv:2207.11280 [cs]
[13] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2023. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? https://doi.org/10.2139/ssrn.4496380
[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs]
[15] Spandan Garg et al. 2022. DeepDev-PERF: A Deep Learning-Based Approach for Improving Software Performance. In *Proceedings of the 30th ACM Joint ESEC/FSE*. 948–958. https://doi.org/10.1145/3540250.3549096
[16] Spandan Garg et al. 2023. RAPGen: An Approach for Fixing Code Inefficiencies in Zero-Shot. arXiv:2306.17077 [cs]
[17] Dan Hendrycks et al. 2021. Measuring Coding Challenge Competence With APPS. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* 1 (Dec. 2021).
[18] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. arXiv:1904.09751 [cs]
[19] Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. 2023. Stop Uploading Test Data in Plain Text: Practical Strategies for Mitigating Data Contamination by Evaluation Benchmarks. arXiv:2305.10160 [cs]
[20] Kevin Jesse et al. 2023. Large Language Models and Simple, Stupid Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 563–575. https://doi.org/10.1109/MSR59073.2023.00082
[21] Raymond Li et al. 2023. StarCoder: May the Source Be with You! (2023). arXiv:2305.06161 [cs.CL]
[22] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. 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).
[23] Shuai Lu et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs]
[24] Ziyang Luo et al. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs]
[25] Aman Madaan et al. 2023. Learning Performance-Improving Code Edits.
[26] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5. https://doi.org/10.1145/3524842.3528470
[27] Erik Nijkamp et al. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs]

[12] See https://www.grid5000.fr

[28] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs]
[29] Hammond Pearce et al. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571
[30] Neil Perry et al. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. 2785–2799. https://doi.org/10.1145/3576915.3623157
[31] Ruchir Puri et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* 1 (Dec. 2021).
[32] Baptiste Rozière et al. 2023. Code Llama: Open Foundation Models for Code. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs]
[33] Gustavo Sandoval et al. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2205–2222.
[34] Priyan Vaithilingam et al. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–7. https://doi.org/10.1145/3491101.3519665
[35] Helena Vasconcelos et al. 2023. Generation Probabilities Are Not Enough: Exploring the Effectiveness of Uncertainty Highlighting in AI-Powered Code Completions. (2023). https://doi.org/10.48550/ARXIV.2302.07248
[36] Roberto Verdecchia et al. 2017. Estimating Energy Impact of Software Releases and Deployment Strategies: The KPMG Case Study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 257–266. https://doi.org/10.1109/ESEM.2017.39
[37] Shiqi Wang et al. 2022. ReCode: Robustness Evaluation of Code Generation Models. https://doi.org/10.48550/ARXIV.2212.10264
[38] Yue Wang et al. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs]
[39] Jason Wei et al. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs]
[40] Frank F. Xu et al. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th International Symposium on Machine Programming (MAPS 2022)*. 1–10. https://doi.org/10.1145/3520312.3534862
[41] Weixiang Yan et al. 2022. WhyGen: Explaining ML-powered Code Generation by Referring to Training Examples. In *Proceedings of the 44th Int. Conf. on Software Engineering, vol. 2 (ICSE '22)*. 237–241. https://doi.org/10.1145/3510454.3516866
[42] Burak Yetistiren et al. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the 18th Int. Conf. on Predictive Models and Data Analytics in Software Engineering*. 62–71. https://doi.org/10.1145/3558489.3559072
[43] Hao Yu et al. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. arXiv:2302.00288 [cs]