Performance Comparison of GGUF-Formatted LLMs on Code Completion and Suggestion

SOEN691: Final Report

Peter Sakr
40237311
Concordia University
Montreal, Canada
peter.l.sakr@gmail.com

Priyadarshine Kumar
40293041
Concordia University
Montreal, Canada
priyadarshine.2322@gmail.com

Divya Divya
40190738
Concordia University
Montreal, Canada
ds.divya268@gmail.com

Soham Mukherjee
40190735
Concordia University
Montreal, Canada
sohammukherjee1996@gmail.com

ABSTRACT

This report summarizes the research and analysis that is conducted in our project. We begin by explaining several tasks that LLMs are used in, as well as several popular LLMs that specifically have been trained on coding tasks. We explain the problem of heavy resource requirements that is required during both training as well as inference. Our study focuses on GGUF-formatted models to assess their performance and feasibility compared to non-quantized models. We conduct an empirical study to evaluate their performance in several tasks involving code generation and completion as well as assessing the difference in performance based on the selected quantization level. We also study the generalizability of these models across several programming languages. The results indicated that GGUF models are feasible replacements to traditional models, especially when using a system with limited memory resources. Furthermore, the results seemed to generalize among different languages that may have different distribution in the training data.

1 Introduction

1.1 Context

Large Language Models (LLMs) have significantly impacted the AI community, proving their utility across various natural language tasks [1]. These models have evolved to include specialized functions within the software engineering domain, notably in tasks like code generation and code completion. Code generation involves translating a natural language prompt into functional code, while code completion offers contextual suggestions to complete a given code snippet. Such capabilities are integrated into development environments and code editors to assist programmers while writing code [2].

In July 2021, OpenAI launched Codex, an LLM trained on public code databases, which was later incorporated into tools like GitHub CoPilot [3]. Codex, derived from the GPT-3 family and with models up to 12 billion parameters, demonstrated its proficiency by solving 28.8% of problems from the HumanEval problem set outright and 70.2% with multiple samples per problem [3].

Following Codex, OpenAI introduced GPT-4 in March 2023. This model, trained in vision, text and code, improved upon its predecessor with a success rate of 67% on the same HumanEval set [4]. This advancement marks a significant step in the evolution of coding related LLMs.

1.2 Motivation

The motivations for developing and refining LLMs like Codex and Code Llama stem from a desire to embed AI deeper into the software development lifecycle [2]. This includes roles in requirements engineering, design, development, quality assurance, maintenance, and management. The integration of AI aims to enhance productivity, reduce error rates, and streamline the development process, making it less cumbersome and more intuitive for developers.

Meta's Code Llama, launched in mid-2023 with model sizes ranging from 7 billion to 70 billion parameters, highlights this competitive landscape. These models, freely available, demonstrate capabilities comparable to GPT-4 across various problem sets, including HumanEval [5]. This reflects a broader trend in the AI community: a push towards creating more sophisticated, efficient, and accessible tools for software development, underlining the deepening intersection of AI technology with practical software engineering solutions.

The introduction of GPT-Generated Unified Format (GGUF) and its evaluation using the Big Code Model Leaderboard aligns with these developments, offering a glimpse into the potential for more computationally and memory-efficient models that could reshape how LLMs are utilized in software development tasks.

1.3 Problem

The research and development phase of LLMs has shown to require an intense amount of compute resources, which in turn have a large effect on the sustainability of LLM development when considering the required carbon footprint. Furthermore, it was shown by researchers at Facebook that, depending on the task at

hand, ML inference (usage after model deployment) could require a cumulative resource requirement greater than what was required during the research and development phase [6]. One important resource cost while using an LLM is the amount of memory required to store the model's weights during inference. According to Code Llama's file sizes on huggingface.co, it requires at least 13.5 GB of memory to store the weights of the smallest model in its family.

To reduce the resource requirements of LLMs, a team of researchers at Google investigated the feasibility of using a larger, already-trained model to fine-tune a pre-trained compact architecture. The results of this research showed that fine-tuned and pre-trained compact models could be improved further using standard transfer learning techniques throughout the fine-tuning phase [7]. Another, more feasible, solution that has been thoroughly researched is the idea of compressing and quantizing large models to require less compute and memory during inference. Initially, work was done to quantize the numeric precision of model parameters as well as the activation function in the hopes of reducing the overall size of the model while only reducing performance an acceptable amount [8]. Different quantization types were also studied, with INT4 and INT8 quantization being initially introduced in the literature, and FP8 introduced more recently. Furthermore, research was done into quantizing parameter weights and activation functions differently from each other to increase performance while keeping storage efficiency high [8].

With the rise of LLM research in the open-source community, some libraries were able to help with quantizing and distributing modern LLM models. A major step was the creation of Llama.CPP[1] which allows running inference of major models using only C/C++ on local CPU and GPU hardware. It was made possible using the ggml library[2], which was made by the same developer, which created a standardized model format that allowed for different quantization types as well as conversion from popular model formats like PyTorch and TensorFlow. This format is currently called the GGUF (GPT-Generated Unified Format[3]) format. Some of its predominant features include supporting the major machine learning libraries (PyTorch and Tensorflow), being a single file containing all the necessary weights and metadata to run the model and supporting many different types of quantizations and quantization levels.

As a result of this development, a 13 billion parameter model could theoretically run on hardware with just 8GB of RAM. While a lot of research has been done to compare code generation and completion models with less parameters or that used fine-tuning, we would like to investigate the performance of models that are quantized and compressed in the GGUF format to check whether their usage in code generation and completion is feasible if they are run locally, or on a networked server as an alternative to the popular online models such as GitHub Copilot.

2 Methodology

2.1 Research Questions

This study investigates the following three research questions:

1 **Quantization Comparison:** How do different quantization levels affect the performance of locally running LLMs in the GGUF format?

This RQ aims to investigate the impact of varying quantization levels on the performance of GGUF LLMs.

2 **Performance Comparison:** How do large language models compare to GGUF-formatted models in terms of generating correct code completion for given tasks?

This RQ aims to verify whether traditional LLMs exhibit superior performance in generating accurate code completions compared to GGUF models, and how much of a difference in performance there is. This RQ also aims to consider the size difference of the quantized versus unquantized models.

3 **Generalization** Do GGUF-formatted models generalize well across different languages, or does the compression forego performance in different languages?

This RQ seeks to explore the generalization capabilities of GGUF language models across different programming languages, to check if they can be used across different projects and to understand if there is any loss in performance on languages that are less often seen in training data when quantized.

2.2 Study Design

To address these research questions, we have developed an empirical study that will allow us to conduct the necessary analysis. We downloaded and set up GGUF-formatted LLMs to use on our machines, and then ran several benchmarks to collect performance information. To make the study simpler, we will only consider models that (in their quantized and GGUF format) require less than or equal to 8GB of RAM. We also limited our study to freely available models (CodeLlama 2 and DeepSeek Coder)

For RQ1:

We run benchmarks where the same GGUF LLMs are evaluated at different quantization levels, ranging from low to high precision. We measure performance metrics such as pass@k as well as memory requirements to compare their relative performance.

Specifically, we run the HumanEval benchmark on two LLMs (CodeLlama 2 and DeepSeek Coder) on several quantization levels each. Memory resources are measured based off required RAM and VRAM to run the models on our systems. We then select a 'best'

[1] Available at https://github.com/ggerganov/llama.cpp
[2] Available at https://ggml.ai
[3] GGUF format explanation available at https://github.com/ggerganov/ggml/blob/master/docs/gguf.md

quantization level that considers the smallest model size for the least amount of performance loss.

For RQ2:

We run benchmarks on the GGUF LLMs (at the 'best' quantization level) where they are tasked with completing code snippets provided with natural language prompts or code snippets. We utilize the same standardized evaluation metrics based on the specific benchmark such as pass@k. We then compare the obtained results to a baseline, which would be standard LLMs performance using the same benchmark, as found in the Big Code Model Leaderboard[4]. More specifically, we compare the performance difference as well as the size difference to attempt to conclude the feasibility of GGUF models when resources are scarce. We also make sure to use the same model parameters as those shown on the leaderboard to reduce the potential for any bias.

For RQ3:

These experiments will involve GGUF LLMs being evaluated on benchmarks containing similar tasks from multiple programming languages. The collected metrics (such as pass@k) would allow us to compare the performance of the same LLM across several programming languages, thus allowing us to analyze whether these LLMs can be generalized to work in different types of projects. We look at different programming languages, some mentioned to be part of the training data, and others not, to be able to understand the difference in performance.

2.3 Benchmarks

While exploring the literature on code generation and completion LLMs we have noticed several benchmarks that were referenced and used to evaluate their performance. To make sure we would have an accurate baseline, we decided to explore and utilize these same benchmarks to have comparative results. The benchmarks mentioned in the literature are:

- **HumanEval:** HumanEval was developed by OpenAI and was used to benchmark the performance of Codex. It contains 164 Python problems as well as unit-tests to measure functional correctness. Since functional correctness is being measured, pass@k is used as the metric [3].
- **MBPP (Minimalistic Basic Python Problems)**: This benchmark contains 974 short python problem prompts that evaluate code generation capabilities [9]. It also analyzes functional correctness and uses pass@k.
- **MultiPL-E:** is a benchmark to analyze an LLMs code generation capabilities that supports 18 different programming languages [10]. The authors developed it based off the two other benchmarks HumanEval and MBPP.

Thus, for RQ1 and RQ2, we were hoping to make use of HumanEval as well as MBPP to keep all performance comparisons on the same programming language. However, due to time constraints in how long the models require to run a benchmark, we have decided to only use HumanEval. Finally, when dealing with RQ3 we use MultiPL-E to assess the generalizability of the results across several different languages.

2.4 Pass@k Metric:

When investigating the metrics that allow comparison among different runs, pass@k seemed to be the most promising, as it can be used to directly compare the functional correctness between different models. Pass@k refers to the percentage of samples among k samples that 'pass' the tests for that specific problem. However, this simple version of pass@k (ratio of number of correct samples to the number of samples k) does suffer from a bias due to the reproducibility issues that LLMs suffer from [3]. As a result, Chen et al. [3] have developed an unbiased version for this metric that reduces the possibility in having a large variation when run several times. Equation 1 shows the mathematical notation of this metric. The variables are as follows: n is the number of total samples retrieved from the LLM, c is the number of correct samples, and k is the number of samples being looked at (k < n).

$$\textbf{pass@k} = \mathbf{E}_{\textbf{problems}}\left[\mathbf{1} - \frac{\binom{\mathbf{n-c}}{\mathbf{k}}}{\binom{\mathbf{n}}{\mathbf{k}}}\right]$$

Equation 1: unbiased pass@k

Chen et. al also supplied a simple Python snippet that can calculate this same metric [3]. This code snippet is shown in figure 1.

```
def pass_at_k(n, c, k):
    """
    :param n: total number of samples
    :param c: number of correct samples
    :param k: k in pass@$k$
    """
    if n - c < k: return 1.0
    return 1.0 - np.prod(1.0 - k /
        np.arange(n - c + 1, n + 1))
```

Figure 1: code snippet to calculate unbiased pass@k

2.5 Other Used Tools

We have relied on two main software tools to run the GGUF models, which are LM Studio and ollama. Both of these tools allow a GGUF model to be run on local hardware (with optional GPU acceleration, if available) and creates a local OpenAI-compatible API endpoint to interact with the model.

Since GGUF models are usually provided by the community on huggingface.co, we decided to use models converted and quantized by the same individual (TheBloke user on huggingface) to limit the

[4] Big Code Model Leaderboard found here

Table 1: HumanEval Results for several quantization levels

Model	Quantization	Model Size (GB)	HumanEval Result		
			Pass@1	Pass@10	Pass@100
CodeLlama	Q3	3.6	13.97%	35.28%	53.04%
	Q4	4.08	27.13%	45.89%	65.89%
	Q6	5.53	26.24%	44.74%	62.13%
	Q8	7.16	25.88%	43.94%	60.35%
DeepSeek Coder	Q3	3.6	33.84%	62.27%	74.28%
	Q4	4.08	37.33%	63.19%	74.61%
	Q6	5.56	40.49%	64.44%	75.99%

possibility of any potential differences that could be created because of different quantization tools.

To run the benchmarks, we relied on an evaluation harness created by the BigCode community[5]. This is essential to make sure we use the same evaluation methods that created the results on the Big Code Model Leaderboard. However, several small changes needed to be made to be able to call models that are served from a local API endpoint[6].

3. Results

3.1 Research Question 1

Table 1 summarizes the results of several runs of HumanEval. Each run was performed on a different quantization of two models: CodeLlama and DeepSeek Coder. When sampling the LLM for each prompt, a temperature of 0.2 was used, which matches what online resources suggest for the HumanEval benchmark. The number of samples n for the pass@k metric was selected as 200, which is what many articles in literature choose to use, including Chen et al [3]. As a result, pass@1, pass@10 and pass@100 were calculated from these samples. For each model, the line with the bolded results indicates the highest scores.

The model size was also reported here to be able to draw conclusions based on resource consumption. We were able to note that both ollama and LM Studio required a constant amount of memory directly related to the model size. More specifically, both programs required an overhead of around 800MB (on top of the model size) to run the OpenAI-compatible server endpoints. This total memory requirement is split between RAM and VRAM depending on the user's settings in choosing the number of layers of the model that should be run on the GPU. If the entire model is run from the GPU, the RAM requirement would be the 800MB overhead, and the GPU would store the model.

For both models, we performed the analysis on the Q3, Q4 and Q6 quantizations. We also checked the Q8 quantization for CodeLlama. These symbols refer to the number of bits that represent each parameter (eg: Q4 represents a 4-bit quantization format).

We noticed that the quantization level does have a direct impact on the performance of the model, however, the extent of this impact seems to decrease as we go beyond Q4. The graph of figure 2 may make this analysis easier to visualize.

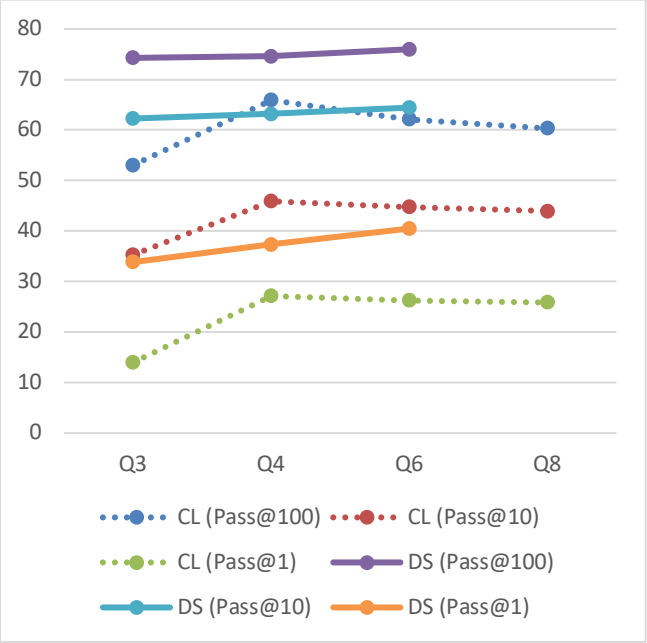

Figure 2: HumanEval Results in a line chart

In fact, Q4 seems to have the highest performance for CodeLlama, and both Q6 and Q8 quantizations lower the performance relative to Q4. This result is not consistent with DeepSeek's. Q6 quantization shows the highest HumanEval scores, and Q8 could potentially increase this further. Thus, we can easily conclude that different quantization levels do affect model performance;

[5] Original repository available here

[6] Repository with required changes available here

however, the effect is different from model to model, and the 'best' quantization level in terms of performance depends on the model being used.

For the rest of our research questions, we aimed to use the quantization level for each model that was able to save the most memory space, while still retaining 'good enough' performance. We wanted to see if we were able to find a single quantization level among both models that was able to fulfill this requirement, or if this also depended on the model.

Table 2: Size and Score Increase relative to Q3

Model Info		Size Increase	Pass@1 Increase
CodeLlama	Q4	13%	94%
	Q6	54%	88%
	Q8	99%	85%
DeepSeek Coder	Q4	13%	10%
	Q6	54%	20%

Table 2 summarizes the relative increase in both model size, as well as the Pass@1 score among different quantization levels for each model, relative to Q3 of that model. In CodeLlama we notice that Q4 has the smallest size increase and the largest pass@1 increase, which is obvious since Q4 had the highest pass@1 score. However, looking at DeepSeek Coder, we noticed that Q4 resulted in a 10% better score, with only a 13% size increase. Q6 would result in a better overall performance due to a 20% increase, however, it would also result in a model that is 54% larger. Thus, even when looking at DeepSeek Coder, Q4 seems to be the most efficient in terms of both performance and model size.

3.2 Research Question 2

To compare the results of the Q4 versions of each model to those reported in the Big Code Leaderboard, we run HumanEval again, but with n = 50 and a temperature of 0.2 to match the specifications given in the leaderboard information section.

The results for both CodeLlama and DeepSeek Coder are shown in Table 3. The results for the unquantized models are directly retrieved from the leaderboard and thus the pass@10 results are missing simply because they are not reported on the leaderboard.

Table 3: HumanEval Results for quantized and unquantized models

Model	Quantized?	HumanEval Results	
		Pass@1	Pass@10
CodeLlama	Yes (Q4)	26.88%	44.76%
	No	29.98%	-
DeepSeek Coder	Yes (Q4)	37.20%	62.78%
	No	45.83%	-

As expected, the unquantized models have a higher performance than the quantized alternatives. However, the loss in performance that the quantization process results in seems to be very slight, with a 10.34% decrease in CodeLlama's performance and a 18.83% decrease in DeepSeek Coder's performance. To further investigate this issue, we also accounted for the relative size difference (ie the amount of memory required to load and run the model) in our analysis. Table 4 shows the difference in size for the models.

Table 4: Size Difference between quantized and unquantized models

Model	Quantized?	Model Size (GB)	Size Difference
CodeLlama	Yes (Q4)	4.08	-69.73%
	No	13.48	
DeepSeek Coder	Yes (Q4)	4.08	-69.73%
	No	13.48	

The sizes for the quantized models were retrieved by adding the sizes of their saved weights on Huggingface. When taking size into account, we notice that the quantized models have a size that is almost 70% smaller (both models have the same number of parameters and thus share the same size in both formats). As a result, we can note that quantization does result in a 10%-20% decrease in performance, however, it also saved around 70% of available memory space.

3.3 Research Question 3

While the authors of CodeLlama did not mention the distribution of programming languages in the training dataset, they noted that the dataset was predomninantly a "near-duplicated dataset of publicly available code" [5]. As such, we assume that the dataset consists of a distribution similar to the distribution in public repositories such as on GitHub. As a result, we assumed that Python, Java and C++ were among the languages most often found, and PHP, Shell and GOLang were among the lesser seen

Table 5: Multi-PLe results for PHP, JS and Shell

Quantized?	PHP		JS		Shell	
	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10
Yes (Q4)	17.87%	31.86%	19.84%	34.03%	9.09%	24.22%
No	25.17%	-	31.80%	-	12%*	-

languages by the model during training. We selected these languages, in particular, for a few reasons. First, we wanted to have a selection of both widely used languages and languages that had a lesser usage to understand the impact of quantization on them. We also selected these since we were able to find baselines for them either directly from the literature (ie [5]), or from the Big Code Leaderboard. Furthermore, all of these languages are available under the Multi-PLe benchmark mentioned earlier.

For all of the following results, we evaluated CodeLlama (Q4) with a temperature of 0.2 and a sample of n per prompt. We also only relied on 50 prompts from the dataset to reduce the amount of time needed to generate the results for all the different languages. The following table shows the results for CodeLlama for both Java and C++. The baseline was copied from the Big Code Leaderboard.

Table 6: Multi-PLe results for Java and C++

Quantized?	Multiple-Java		Multiple-CPP	
	Pass@1	Pass@10	Pass@1	Pass@10
Yes (Q4)	24.53%	42.25%	24.70%	42.93%
No	29.20%	-	27.23%	-

The results indicate a very similar loss as seen in the results of HumanEval when comparing the quantized to the unquantized model. More specifically, the Java results show a loss in performance of 15.9% and C++ exhibits a loss of 9.3%.

Table 5 shows the results for JavaScript, PHP and Shell (Bash). These are 3 of the 4 languages we selected to represent the data that could be underrepresented in the training dataset. The asterisk next to the baseline for the Shell language is only to note that this result was copied from [5] rather than the Big Code Leaderboard. Interestingly, Shell exhibits a loss of 24.5%, JS indicates a loss of 37.6% and PHP a loss of 29%. The difference in loss could indicate that different languages may behave differently.

We also ran Multi-PLe on one more language, which is GoLang. Unfortunately, GoLang was never evaluated for the unquantized version of CodeLlama and thus we do not have a baseline to compare our result to. Interestingly, this language also exhibited a score that we had not seen among the previous ones. GoLang resulted in a pass@1 score of 76.16% and a pass@10 score of 97.72%. At first glance, this may indicate that CodeLlama is far superior in applications involving Go, however, it may also indicate that the model was overfitting on Go, or that the specific tasks involved here were in the training dataset.

4. Discussion

Overall, the collected data and results show that quantized models in the GGUF format are quite feasible in terms of overall performance. Compared to traditional models, they do suffer from a bit of loss in performance; however, depending on the desired quantization, this loss could be minimal. The results of RQ1 did show that the decrease in performance among quantization levels of the same model does differ based on the desired model. As a result, no immediate conclusion could be made to always choose a specific quantization level. However, our research showed that when model size is also analyzed and taken into account, Q4 (ie 4-bit quantization) seemed to have the best performance relative to its comparatively small size.

Furthermore, when comparing a quantized model to its non-quantized alternative in RQ2, the actual difference in performance may not remain the same if looking at multiple different models. This was indicated when we saw a performance difference of roughly 10% for one model, but 18% for another. However, even an 18% loss in performance may be manageable if we consider that it also saves 70% of memory and storage requirements for the model. Thus, depending on the circumstance, quantization could be an ideal solution such as when dealing with machines with limited amount of memory. On the other hand, a larger company with a large budget working on client-facing software may wish to forego the storage saving if they need a more performant model.

The analysis of RQ3 has shown that the performance of a model across several programming languages is usually similar to the analyzed performance of Python as in RQ2. More often than not, the model itself plays a role here rather than being able to conclude that the quantization itself affected results. A model that has been trained on more programming languages will still be able to have consistent performance on them even when quantized.

5. Threats to Validity

Our study has several threats to validity, some internal and some external.

Internal threats: While the study design does allow to generalize most of the drawn analysis, some areas could have room for improvement and as a result may not generalize as well.

Specifically, for research question two, we had some differing results in the loss difference, and thus found that a range of 10-20% of loss between quantized and unquantized models is normal. This may not be generalizable and could fully depend on the chosen model. Furthermore, the results of RQ3 could be made to generalize if we had been able to run the benchmarks on multiple models and not just CodeLlama. Also, we recognize that LLMs can exhibit output variability, even under consistent input conditions. This variability can affect the reproducibility and generalizability of our findings. To mitigate this, we ensured that multiple runs were conducted for each test case, allowing us to use statistical methods to understand and report the range and distribution of our results. Moreover, the performance of quantized models may also be influenced by the specific hardware and software environment in which they are run. We controlled this by standardizing the testing environment across all models to the greatest extent possible.

External threats: The used baselines are all based on external sources. Those referenced from [5] do not have any specifics in terms of model parameters. Furthermore, the results from Big Code Leaderboard are not reviewed by researchers and may be invalid to consider as a baseline.

6. Conclusion

This study has overseen the analysis of GGUF formatted quantized LLMs, specifically in the use case of code generation. Our experiments have shown that GGUF models have different performance based on the quantization level, and that a 4-bit quantization may have good enough performance for a significant decrease in size. Furthermore, when compared to unquantized models, they have lesser performance, but GGUF models use much less memory and thus could be great alternatives when dealing with low-resource environments. Finally, these models have shown that they exhibit similar behavior to unquantized models when generating code in different programming languages.

The work in this study opens the road to several avenues for future research. First and foremost, GGUF models could be examined in more depth in other areas of Software Engineering, not just code generation. In addition, studies trying to understand when a specific quantization level should be used could also be investigated. Finally, work could be done to analyze different quantization formats in general to see if the GGUF format has negative effects that could be reduced.

REFERENCES

[1] W. X. Zhao *et al.*, "A Survey of Large Language Models." arXiv, Nov. 24, 2023. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/2303.18223

[2] X. Hou *et al.*, "Large Language Models for Software Engineering: A Systematic Literature Review." arXiv, Sep. 12, 2023. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/2308.10620

[3] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code." arXiv, Jul. 14, 2021. Accessed: Feb. 13, 2024. [Online]. Available: http://arxiv.org/abs/2107.03374

[4] OpenAI *et al.*, "GPT-4 Technical Report," 2023, doi: 10.48550/ARXIV.2303.08774.

[5] B. Rozière *et al.*, "Code Llama: Open Foundation Models for Code," 2023, doi: 10.48550/ARXIV.2308.12950.

[6] C.-J. Wu *et al.*, "Sustainable AI: Environmental Implications, Challenges and Opportunities." arXiv, Jan. 09, 2022. Accessed: Mar. 07, 2024. [Online]. Available: http://arxiv.org/abs/2111.00364

[7] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-Read Students Learn Better: On the Importance of Pre-training Compact Models." arXiv, Sep. 25, 2019. Accessed: Mar. 07, 2024. [Online]. Available: http://arxiv.org/abs/1908.08962

[8] H. Shen, H. Chang, B. Dong, Y. Luo, and H. Meng, "Efficient LLM Inference on CPUs." arXiv, Dec. 07, 2023. Accessed: Mar. 07, 2024. [Online]. Available: http://arxiv.org/abs/2311.00502

[9] J. Austin *et al.*, "Program Synthesis with Large Language Models." arXiv, Aug. 15, 2021. Accessed: Mar. 07, 2024. [Online]. Available: http://arxiv.org/abs/2108.07732

[10] F. Cassano *et al.*, "MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 7, pp. 3675–3691, Jul. 2023, doi: 10.1109/TSE.2023.3267446.