ISBN: 978-981-94-1156-6
2024 the 14th International Workshop on Computer Science and Engineering (WCSE 2024)
doi: 10.18178/wcse.2024.06.042

Beyond Synthetic Benchmarks: Assessing Recent LLMs for Code Generation

Amirkia Rafiei Oskooei [1,2], Mehmet Semih Babacan [2], Erdi Yağcı [2],

Çağatay Alptekin [2] and Ali Buğday [2 +]

[1] R&D Center, Intellica Business Intelligence Consultancy, Istanbul, Turkey

[2] Computer Engineering Dept., Yildiz Technical University, Istanbul, Turkey

Abstract. Large Language Models (LLMs) have garnered significant attention for their diverse capabilities across various applications. This study delves into their potential for code generation, a task witnessing the emergence of specialized LLMs. However, evaluating these models remains a challenge. This work proposes a novel approach to LLM evaluation in code generation. We leverage a dataset of real-world programming tasks culled from university student homework assignments, offering a window into practical programming experiences. We assess recent LLM models using informal natural language prompts formulated by non-native English speakers, reflecting the variety of user inputs encountered in practice. Our findings reveal that while these LLMs demonstrate promise, their success rates in solving problems on the first attempt remain modest, especially for more complex tasks. This highlights the need for continued research in fine-tuning LLMs for code generation. Overall, this study contributes to the field by introducing a unique evaluation methodology employing real-world problems and natural language prompts, offering valuable insights into the current capabilities and limitations of LLMs in code generation.

Keywords: Large Language Models, Code Generation, Natural Language, Program Synthesis, LLM Benchmark, Generative AI

1. Introduction

Large Language Models (LLMs) have captured significant interest in recent years due to their remarkable capabilities across a wide range of applications. These models excel in tasks like text generation, translation, and question answering, demonstrating their potential to revolutionize various industries [1]. Within the software engineering domain, LLMs are making a substantial impact, particularly in the realm of code generation [2]. Initially, LLMs were explored for general-purpose tasks that could include code generation. However, the surge in interest for program synthesis using LLMs has spurred the development of specialized models designed and fine-tuned specifically for code generation [3]. As the number of these models continues to grow, the need for robust evaluation and benchmarking methodologies becomes increasingly crucial.

Existing research has explored various approaches to evaluate LLMs for code generation. Some studies focus on function-level or class-level code generation, while others utilize datasets from diverse sources such as programming contests, manually written benchmarks, or even test the models for functionalities beyond code generation [4]. Despite the valuable findings and robust methods employed in these studies, limitations remain to be addressed. Firstly, many existing studies rely on synthetic benchmarks that may not accurately reflect the real-world challenges faced by programmers. Secondly, the prompts used to test LLMs often consist of pre-processed, isolated elements like function signatures or natural language text, failing to capture the variety of inputs encountered by these models in real-world scenarios with millions of users [5] [6]. Most importantly, the rapid evolution of this field, with new models emerging at an accelerated pace, presents a significant challenge. Keeping pace with this constant innovation in terms of evaluation and benchmarking demands a substantial investment of time and computational resources [7].

Our study aims to contribute to this field by evaluating recent LLMs using a real-world dataset of programming tasks collected from university student homework assignments in computer engineering. We

+ Corresponding author.
E-mail address: {semih.babacan, erdi.yagci, cagatay.alptekin, ali.bugday}@std.yildiz.edu.tr; amirkia.oskooei@intellica.net.

distinguish our work by leveraging informal natural language prompts written by non-native English speakers, without relying on specific pre-defined formats. This approach is particularly relevant as LLMs may struggle to accurately interpret user intent when presented with formal specifications. Additionally, our dataset reflects diverse, realistic, and practical use cases gleaned from computer engineering homework assignments, providing a more accurate representation of the problems programmers encounter in practice. Ultimately, our research complements previous studies by evaluating some of the latest models, such as llama3:8b. This focus on cutting-edge models is crucial in this fast-paced domain, where constant assessment using diverse benchmarks and approaches remains a critical necessity for advancing the field of LLM-based code generation.

2. Literature Review

Recent research delves into the exciting potential of Large Language Models (LLMs) across various software engineering tasks. Beyond the much-explored realm of code generation, studies investigate LLM capabilities in detecting and correcting errors (bugs) within code, automating code documentation generation, and even aiding program comprehension. These efforts paint a promising picture of LLMs becoming valuable tools throughout the software development lifecycle [8]. However, a significant portion of research centers specifically on evaluating LLMs for their code generation capabilities. Existing approaches delve into LLM proficiency at various code creation levels. Some studies focus on class-level generation [4], where the LLM takes a high-level problem description and attempts to generate entire software classes. In contrast, other research explores function-level completion [6], where the LLM takes partially written code and fills in the missing functionality to achieve the desired outcome. Additionally, researchers have investigated using LLMs to generate test cases that act as a safety net, validating the correctness of the code they produce [9]. Beyond code functionality, researchers are investigating the role of LLMs in enhancing code quality through automatic documentation generation [10]. Here, the LLM analyzes the written code and creates documentation that explains its purpose and functionality. Perhaps the most ambitious research area involves LLM-based bug fixing [11]. In this domain, the model analyzes existing code, identifies errors, and proposes potential fixes. This capability, if perfected, could significantly improve software development efficiency and reliability.

Several benchmarks have emerged to assess LLM performance in code generation, each offering unique strengths and considerations. HumanEval [6], a foundational benchmark, features 164 hand-written Python problems designed to evaluate a model's understanding of language, reasoning, algorithms, and basic mathematics. Building upon this, Multi-HumanEval [12] expands the evaluation scope by encompassing over 10 programming languages. It achieves this by leveraging a scalable conversion framework that translates prompts and test cases from the original HumanEval dataset into the target languages. CodeContests [13] focuses on Python and C++ code generation by employing 165 problems sourced from Codeforces, a platform for real-world coding competitions. Moving beyond synthetic problems, some benchmarks incorporate more realistic datasets. DS-1000 [14], for example, comprises 1000 statement-level tasks collected from StackOverflow, reflecting the types of coding challenges programmers encounter in practice. Similarly, SweBench [15] and CoderEval [16] leverage datasets curated from popular open-source projects hosted on GitHub. SweBench boasts 2,294 software engineering problems extracted from real GitHub issues and pull requests across 12 prominent Python repositories. CoderEval, on the other hand, provides a collection of 230 Python and 230 Java code generation tasks. The choice of benchmark significantly impacts the interpretation of LLM evaluation results, with each offering valuable insights into different aspects of a model's code generation capabilities.

While much of the cited research has concentrated on particular computational and technical approaches, this article primarily addresses the use of language models for code creation. As an example, in big data initiatives, the authors of [22] focus on improving user interface testing by analyzing graph data. On the other hand, the language models are the main focus of our research. While this work explores the use of big language models mostly in code generation, [23] covers the structural code clone detection issue, which is more in line with research in software engineering and code analysis. While this work focuses on language models, the authors of [24] address a different aspect of data processing and administration, demonstrating the many applications of data technology. Contrasting with the investigation of big language models for code generation, the authors of [25-26] and [27-28] place an emphasis on data representation and user behavior analysis,

demonstrating the interdisciplinary character of modern computational research. Contrasting with the language models study carried out in this study, the authors of [29-30] talk about collaborative grid services, with an emphasis on infrastructure and computational support. This highlights the depth of research within the area of computer science and technology. While this work does touch on representation methods for historical data (see [31–32]), its primary emphasis is on code creation using language models.

This work builds upon this existing research by introducing a unique dataset of real-world problems encountered by university students in their homework assignments. By leveraging natural language descriptions as input for the LLMs, our study delves deeper into their ability to comprehend and generate code based on problem statements encountered in educational settings. Furthermore, the rapid evolution of the LLM landscape necessitates ongoing evaluation of new models. Our study contributes to this ongoing process by examining the performance of contemporary LLMs in a novel scenario, complementing the existing literature with fresh data and perspectives.

3. Methodology

This section delves into the methodological details of our study, encompassing the dataset utilized, the Large Language Models (LLMs) evaluated, the design of prompts for the LLMs, the generation of test cases, and the chosen evaluation metrics.

To bridge the gap between theoretical benchmarks and real-world application, we constructed a dataset of 95 programming tasks culled from university student homework assignments at Yildiz Technical University. These tasks represent practical programming challenges encountered by students and are categorized into difficulty levels (Basic, Intermediate, Advanced). The subject areas covered encompass a broad spectrum of programming fundamentals, including Data Manipulation, String Manipulation, Mathematics, Numerical Methods, Computational Mathematics, Data Structures, and Sorting Algorithms. It's important to distinguish that these tasks are not complete source code or classes, but rather the core functions that form the crux of the homework problems. Table 1 provides a comprehensive breakdown of our dataset composition. We opted to evaluate the performance of five open-source (open-weight) LLMs: gemma:2b, gemma:7b, llama2:b, llama3:8b, and mistral:7b [17-19]. These models were chosen due to their recent development, reflecting the fast-paced evolution of the LLM landscape. This selection emphasizes the importance of assessing the capabilities of contemporary models on real-world problems encountered by students.

Table 1. Details of Our Dataset Classified by the Problem Subject

Problem Class	Basic	Intermediate	Complex
Data Manipulation	4	2	1
Mathematics	3	2	5
String Manipulation	5	6	5
Numerical Methods	4	7	6
Data Structures	5	5	5
Computational Mathematics	5	5	5
Sorting Algorithms	5	5	5

Each LLM received a natural language prompt crafted by non-native English speaking university students. These prompts served as instructions, outlining the problem description, the desired programming language (Python in this study), and the function name intended to solve the problem. The generated code should correspond to a function that accepts specified parameters and returns outputs that align with the problem description. To rigorously assess the generated code, we devised a test case generation strategy. Each generated function was presented with specific input values, and the resulting outputs were meticulously compared against pre-defined expected outputs. To mitigate the potential for bias due to minor formatting discrepancies, human evaluators reviewed the results. In some instances, the generated code's logic and output might be functionally correct but exhibit slight formatting variations compared to the expected output. Such discrepancies could lead to failed test cases despite the code's functionality. Our primary evaluation metric

centers on the number of tests passed or failed by each LLM for the tasks corresponding to their designated programming language. It's noteworthy that our methodology emphasizes the number of tests passed by each LLM in their initial attempt. While subsequent attempts with additional context (e.g., runtime error messages, human-provided hints) could potentially lead to corrections, this study restricts the LLMs to a single attempt per task to gauge their initial code generation proficiency. In simpler terms, we use the widely-used ***Pass@k*** metric, which calculates the percentage of problems solved based on ***k*** code samples generated for each task:

$$\text{Pass@k} = E_{\text{problems}}\left[1 - \frac{C_{n-c}^{k}}{C_{n}^{k}}\right] \quad (1)$$

In Equation 1**, *n*** represents the total number of samples, ***c*** denotes the number of correct samples, and ***k*** stands for ***k*** in ***Pass@k***.

4. Implementation and Results

To interact with the LLMs and generate code, we employed their official models available through the Hugging Face platform. These models were then deployed within a Google Colab environment. Our Colab runtime configuration leveraged an A100 GPU with 40GB of VRAM, providing sufficient resources for the chosen LLMs. This configuration ensured smooth execution and code generation throughout the testing process. For the LLM configuration, we opted for the default hyperparameters provided by the official models' repo at Huggingface. Our approach to code generation employed greedy sampling, which involves generating a single code sample per LLM. To achieve this, the ***do_sample*** hyperparameter was set to False, effectively setting the ***temperature*** hyperparameter to 0 and promoting the selection of the most likely code sequence. Following code generation, the resulting functions were stored in a structured table designed to facilitate the testing process. This table schema shown at Table 2 encompassed all the essential information required for testing, including the input values, the expected output, and the generated code itself.

Table 2. Detailed Description of the Table Schema

Column Name	Description
Prompt	Description of the test case for the given problem.
Function Name	The name of the function being tested.
Generated Function	The function generated as a result of the test.
Class	Subject classification or category of the function.
Level	Complexity or difficulty level of the function.
Input	Input data used for testing the function.
Output	The actual output generated by the function during the test.
Expected	The expected output for the test case.
Result	The result of the test case, either "Passed" or "Failed".

The final step involved executing the generated functions and calculating the pass rate. Given our employment of greedy sampling with a temperature of 0, we utilized the ***Pass@1*** metric for evaluation. This metric focuses on the success of the first generated code sample produced by each LLM for a particular problem. The detailed results of this evaluation are presented in Table 3.

Table 3. Comparison of *Pass@1* Scores of LLMs, Across Problem Complexity Levels

Level	gemma:2b	gemma:7b	llama2:7b	llama3:8b	mistral:7b
Basic	64.5 %	67.7 %	51.6 %	64.5 %	64.5 %
Intermediate	15.6 %	46.8 %	25 %	40.6 %	37.6 %
Advanced	3.1 %	9.3 %	3.1 %	6.2 %	9.3 %
Overall	27.3 %	41 %	26.3 %	36.8 %	36.8 %

Our evaluation yielded insights into the performance of the chosen LLMs on the real-world dataset of university student homework tasks. The results, presented in Table 3, reveal that gemma:7b achieved the highest overall pass rate across all problem complexity levels (Basic, Intermediate, Advanced). It's important to consider the training data used for each LLM. Both gemma:2b and gemma:7b were trained on 6 trillion tokens encompassing code and mathematical content. In contrast, llama2:7b and llama3:8b were trained on 2 trillion and 15 trillion tokens, respectively. Unfortunately, information regarding the training data for mistral:7b is not publicly available. An intriguing observation lies in the performance of gemma:7b. Despite being trained on a smaller dataset (6T tokens) compared to llama3:8b (15T tokens), it achieved a superior pass rate. This suggests that the architecture of gemma:7b might be more robust for the task of code generation, allowing it to excel even with a less voluminous training dataset. The overall performance of llama3:8b and mistral:7b exhibited a close resemblance. Meanwhile, gemma:2b and llama2:7b demonstrated the lowest pass rates. A possible explanation for llama2:7b's lower performance might be its status as the oldest model among the evaluated ones. Potentially, its training data could be less representative of contemporary programming practices. Gemma:2b outperformed llama2:7b by a slight margin, likely due to its utilization of a newer and more optimized architecture, coupled with a larger training dataset (6T tokens versus 2T tokens). Considering its relatively compact size with only 2 billion parameters, gemma:2b's results appear reasonable.

It's important to acknowledge that, overall, these models did not exhibit exceedingly high success rates on their first attempt at solving the programming tasks, even at the university level (Figure 1).

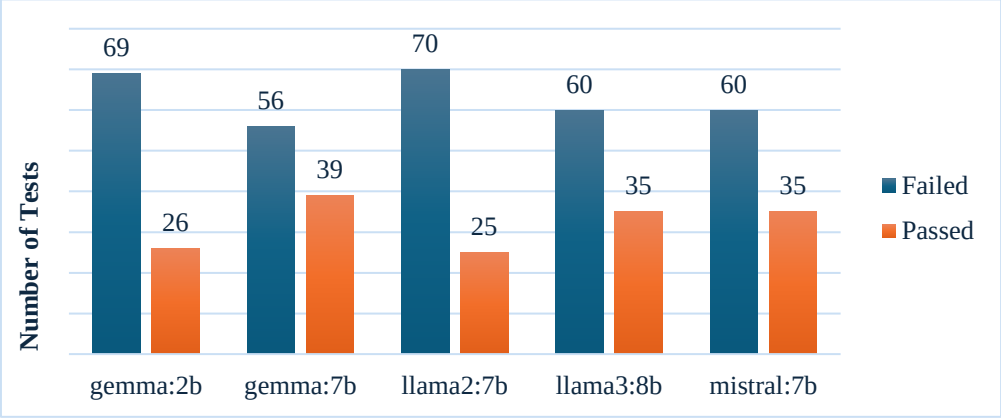

Fig. 1: Summary of Passes and Failures by Each LLM

While they achieved a pass rate of 67.7\% for basic problems, this rate dropped significantly as complexity increased. For intermediate problems, the best model could only solve 46.8\%, and for advanced problems, all models scored below 10\%. These findings highlight the challenges LLMs still face in comprehending natural language descriptions and translating them into functional code, particularly in more intricate scenarios. Furthermore, it's worth noting that larger variation of these models with more parameters likely achieve better performance, but they are not open-source. However, our study, along with similar research efforts, underscores the importance of fine-tuning general-purpose LLMs for specific tasks like code generation. By tailoring these models to the domain of programming, their ability to understand natural language problem descriptions and translate them into accurate code can be significantly enhanced.

5. Conclusion and Future Works

This study investigated the performance of open-source and recently released LLMs in the context of code generation. We leveraged a dataset of real-world programming tasks encountered by university students to assess these models' capabilities. Our findings align with existing research [20-21], highlighting the limitations these models face in generating code accurately on their first attempt, particularly for problems of higher complexity. This underscores the crucial role of fine-tuning general-purpose LLMs for specific domains like code generation.

Several promising avenues exist for future research in this domain. Firstly, the development of even larger and more diverse datasets that incorporate both synthetic and real-world programming challenges is crucial.

Such comprehensive datasets can provide a more robust foundation for training and evaluating LLMs. Additionally, exploring different approaches to test these models can offer deeper insights into their strengths and weaknesses. This could involve employing various evaluation metrics and incorporating human feedback mechanisms into the testing process. As the field of LLM development continues its rapid evolution, with newer, larger models boasting optimized architectures being introduced constantly, the need for ongoing evaluation intensifies. The demand for robust code generation capabilities across various sectors necessitates the continued exploration of how LLMs can be effectively evaluated and tailored to this specific task. This pursuit of effective code generation through LLM evaluation is likely to remain a highly relevant area of research, attracting significant interest from both academic and industrial communities.

6. Acknowledgement

Special thanks to Intellica Consultancy for their significant assistance, vital to the success of this research.

7. References

[1] Kaddour, Jean, et al. "Challenges and applications of large language models." arXiv preprint arXiv:2307.10169 (2023).

[2] Austin, Jacob, et al. "Program synthesis with large language models." arXiv preprint arXiv:2108.07732 (2021).

[3] Nijkamp, Erik, et al. "Codegen: An open large language model for code with multi-turn program synthesis." arXiv preprint arXiv:2203.13474 (2022).

[4] Du, Xueying, et al. "Evaluating large language models in class-level code generation." Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024.

[5] Iyer, Srinivasan, et al. "Mapping language to code in programmatic context." arXiv preprint arXiv:1808.09588 (2018).

[6] Chen, Mark, et al. "Evaluating large language models trained on code." arXiv preprint arXiv:2107.03374 (2021).

[7] Chang, Yupeng, et al. "A survey on evaluation of large language models." ACM Transactions on Intelligent Systems and Technology 15.3 (2024): 1-45.

[8] Jiang, Nan, et al. "Impact of code language models on automated program repair." 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.

[9] Dakhel, Arghavan Moradi, et al. "Effective test generation using pre-trained large language models and mutation testing." Information and Software Technology 171 (2024): 107468.

[10] Su, Yiming, et al. "Hotgpt: How to make software documentation more useful with a large language model?." Proceedings of the 19th Workshop on Hot Topics in Operating Systems. 2023.

[11] Wang, Yue, et al. "Codet5+: Open code large language models for code understanding and generation." arXiv preprint arXiv:2305.07922 (2023).

[12] Athiwaratkun, Ben, et al. "Multi-lingual evaluation of code generation models." arXiv preprint arXiv:2210.14868 (2022).

[13] Li, Yujia, et al. "Competition-level code generation with alphacode." Science 378.6624 (2022): 1092-1097.

[14] Lai, Yuhang, et al. "DS-1000: A natural and reliable benchmark for data science code generation." International Conference on Machine Learning. PMLR, 2023.

[15] Jimenez, Carlos E., et al. "Swe-bench: Can language models resolve real-world github issues?." arXiv preprint arXiv:2310.06770 (2023).

[16] Yu, Hao, et al. "Codereval: A benchmark of pragmatic code generation with generative pre-trained models." Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024.

[17] Team, Gemma, et al. "Gemma: Open models based on gemini research and technology." arXiv preprint arXiv:2403.08295 (2024).

[18] Touvron, Hugo, et al. "Llama 2: Open foundation and fine-tuned chat models." arXiv preprint arXiv:2307.09288 (2023).

[19] Jiang, Albert Q., et al. "Mistral 7B." arXiv preprint arXiv:2310.06825 (2023).

[20] Vaithilingam, Priyan, Tianyi Zhang, and Elena L. Glassman. "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models." Chi conference on human factors in computing systems extended abstracts. 2022.

[21] Liu, Jiawei, et al. "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).

[22] Uygun, Y. et al., On the Large-scale Graph Data Processing for User Interface Testing in Big Data Science Projects, 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 2020, pp. 2049-2056, doi: 10.1109/BigData50022.2020.9378153, 2020.

[23] Kapdan, M. et al., On the Structural Code Clone Detection Problem: A Survey and Software Metric Based Approach. In: , et al. Computational Science and Its Applications – ICCSA 2014. ICCSA 2014. Lecture Notes in Computer Science, vol 8583. Springer, Cham. https://doi.org/10.1007/978-3-319-09156-3 35, 2014.

[24] Pierce, M.E. et al., The QuakeSim Project: Web Services for Managing Geophysical Data and Applications. In: Tiampo, K.F., Weatherley, D.K., Weinstein, S.A. (eds) Earthquakes: Simulations, Sources and Tsunamis . Pageoph Topical Volumes. Birkh¨auser Basel. https://doi.org/10.1007/978-3-7643-8757-0 11, 2008.

[25] Olmezogullari, E., et al., Pattern2Vec: Representation of clickstream data sequences for learning user navigational behavior. Concurrency Computat Pract. Exper. 2022; 34(9):e6546. https://doi.org/10.1002/cpe.6546, 2022.

[26] Olmezogullari, E. et al., Representation of Click-Stream DataSequences for Learning User Navigational Behavior by Using Embeddings, 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 2020, pp. 3173-3179, doi: 10.1109/BigData50022.2020.9378437, 2020.

[27] M. Oz, et al. “On the Use of Generative Deep Learning Approaches for Generating Hidden Test Scripts”, International Journal of Software Engineering and Knowledge Engineering, Vol 31, Issue 10, p1447, 2021.

[28] I. Erdem, et al. “Test Script Generation Based on Hidden Markov Models Learning From User Browsing Behaviors”, 2021 IEEE International Conference on Big Data (Big Data), IEEE Computer Society, 2021.

[29] Nacar, M.A. et al., VLab: collaborative Grid services and portals to support computational material science. Concurrency Computat.: Pract. Exper., 19: 1717-1728. https://doi.org/10.1002/cpe.1199, 2007.

[30] Dundar, Bahadir; Astekin, Merve; Aktas, Mehmet S., A Big Data Processing Framework for Self-Healing Internet of Things Applications, 12th International Conference on Semantics, Knowledge and Grids (SKG), 2016.

[31] Chen, P., Plale, B., Aktas, M.S., Temporal representation for scientific data provenance, 2012 IEEE 8th International Conference on E-Science, e-Science 2012, 2012, 6404477

[32] Aktas, M., Plale, B., Leake, D., Mukhi, N., Unmanaged workflows: Their provenance and use, Studies in Computational Intelligence, 2013, 426, pp. 59–81, 2013.