International Journal of Future Engineering Innovations

Code Generation Using LLMs

Tenneti Lekhya Sri Durga [1*], Mallak Keshava Gayatri [2], Mukku Deepthi Prabha [3], Dr. D Shravani [4]
[1-3] BE, AI&DS, VIII Sem, SCETW, Hyderabad, INDIA
[4] Associate Professor, ADCE dept, SCETW, OU, Hyderabad, INDIA

* Corresponding Author: **Tenneti Lekhya Sri Durga**

Article Info

ISSN (online): 3049-1215
Volume: 02
Issue: 02
March-April 2025
Received: 28-01-2025
Accepted: 24-02-2025
Page No: 16-22

Abstract
Code generation using Large Language Models has brought a significant change in the software industry helping people in various things like understanding the code, writing a code, completion of the code. The main challenge that the Code Generation Models are facing is Hallucinations wherein the given code may not satisfy the users requirements completely or might deviate from the user's requirements. There are many kinds of hallucinations and are divided into different types like the intent conflicting which consists of semantic conflicts, context deviation which includes inconsistency, repetition, dead code. So, to help with the problem of code repetition type of hallucination we proposed a solution that is fine-tuning a pre-trained model on a specific dataset to help in code generation and to reduce the repetition type hallucination. Repetitive code hallucinations occur when the model generates redundant lines increasing complexity and confusing developers. Repetitive code hallucinations remain a key challenge for LLMs. To address this, we curated a dataset of Python code snippets extracted from GitHub repositories. We have also created a User Interface for efficient usability. We made a comparative study with different models and their results. This can further be improvised using prompt engineering, using more vast datasets for finetuning the model for different languages and can be trained to give more accurate and efficient results.

DOI: https://doi.org/10.54660/IJFEI.2025.2.2.16-22

Keywords: Large Language Model(LLMs), Code Generation, Hallucinations, Repetitive Code

1. Introduction

The rapid advancement of technology led to the evolution of Large Language Models (LLMs) which are used in various fields for like explaining concepts, design assistance, giving suggestions, and most importantly generation, debugging, autocompletion of codes making work easier in the field of software development. It has also helped the non-technical people with the development works by generating the codes required by the user giving rise to the code generation models. Despite these advances these models have major challenge that is Hallucinations. The LLMs are trained on vast amounts of data for multiple tasks which makes them general for example we can take a code generation model might be trained by taking vast amount of data including various programming languages which leads to the problem of hallucinations. They remain a persistent challenge in code generation, where the model produces incorrect, redundant, or irrelevant code increasing the lines of code and reducing the readability. This is the concept where the code generated deviates from the user requirements leading to inappropriate results. They are of multiple types like semantic inconsistencies, context deviations, knowledge conflicting. Among these hallucination types the repetition code pose a significant problem reducing the readability and efficiency of the code generated. To address this challenge of repetitive code our contributions are as follows.

- We have developed a fine-tuned model by training it on specific curated dataset of high-quality python codes.
- We have compared the results of our fine-tuned model with other models to understand the effectiveness and efficiency of our model

- Created high quality dataset and fine-tuned the model with different hyper-parameters trying to reduce the repetition of code and increase the efficiency of the code generated.
- Designed a user interface ensuring increase in the ease of accessibility and usability.

The remainder of the paper is organized as follows: Section 2 review related work on different studies and implementations done in the field of code generation using LLMs. Section 3 throws light on the proposed solution. Section 4 presents the experimental results and Section 5 discusses future research directions and concludes the study.

2. Related Work

This section reviews literature on existing works in the field of code generation using LLMs. Fang Liu [1] research is based on the types of hallucinations that are encountered during the code generation using LLMs. They have classified them into different groups. Simiao Zhang [2] research was based on the AI-aided software development prototype that generates use cases**,** system designs, implementations from high level user requirements integrating continuous feedback. Chong Wang [3] here they came up with ToolGen, an approach that integrated autocompletion tools into the code LLM generation process to address the dependencies. Tianyu Wang [4] their study involved categorizing programming questions based on educational requirements, applying various prompt engineering strategies, and assessing the effectiveness of LLM-generated responses. Sarah Fakhoury [5] study based on TiCoder for guided intent clarification through tests for more accurate code generation. Nikhil Parasaram [6] whose study involved in fact selection for generating the appropriate code. They came up with model that selects facts specific to a given bug to include in the prompt. Arun-Balajiee Lekshmi-Narayanan [7] their study involved assessing the feasibility of LLMs to generate code explanations. Daye Nam [8] study is on a UI built directly in the IDE that is geared towards code understanding. Shubham Ugare [9] research involves a novel framework for efficient and general syntactical decoding with LLMs. Wen-Ding Li [10] investigated the ability of LLMs to perform programming by examples where the algorithm is generated from input-output examples.

Juyong Jiang [11] study involves a systematic literature review that serves as a valuable reference for researchers investigating the cutting-edge progress in LLMs for code generation. Shuyuan Xu [12] they developed a novel system for Code Representation and Execution which employs LLM as interpreter to interpret and execute natural language programs. Hagit Gabbay [13] study explores the potential of LLMs to generate feedback on code assignments and to address the gaps in Automated Test-based Feedback. Islem Bouzenia [14] this research introduces RepairAgent to address the program repair challenge through an autonomous agent based on an LLM. Boyang Yang [15] study assesses LLMs repair performance on TutorCode, measuring repair correctness and patch precision. Anupam Garg [16] study involves exploring different prompting strategies, from basic prompts to advanced techniques like chain-of-thought prompting and iterative refinement. Chris Brown [17] their research work conducts a preliminary evaluation exploring the beliefs and behaviors of LLM used to support software development tasks. Federico Cassano [18] comes up with an effective approach for boosting the performance of Code LLMs on low-resource languages using semi-synthetic data. Anisha Agarwal [19] research introduced the Copilot evaluation harness a set of data and tools for evaluating LLM-guided IDE interactions, covering various programming scenarios and languages. Albert Contreras [20] research presents an extensible architecture for the definition of assistance tasks based on LLMs, and their binding to IDE commands and natural language prompts.

Parshin Shojaee [21] research introduced novel approach that leverages the extensive scientific knowledge and robust code generation capabilities of Large Language Models (LLMs) to discover scientific equations from data in an efficient manner. John Chen [22] research paper explores how Large Language Models (LLMs) can assist in Agent-Based Modeling, specifically using NetLogo, a popular programming language for Agent-Based Modeling. Yichen Li [23] research proposed IDECoder a practical framework that leverages IDE native static contexts for cross-context construction and diagnosis results for self-refinement. Julian Van Santen [24] study demonstrated how LLM chatbots can be restricted by teachers to provide a helpful tool for students learning functional programming and its concepts. Yanggyu Lee [25] proposed an effective approach for detecting logical errors with LLMs that makes use of relations among error types in the chain-of thought and tree-of-thought prompts. Irene Weber [26] research provides a taxonomy for LLM integrated applications, offering a framework for analyzing and describing these systems. Dewu Zheng [27] research includes empirical study to deeply understand LLMs code generation performance within settings that reflect the evolving nature of software development. Gregor Jost [28] research paper explores the nuanced impact of informal LLM usage on undergraduate students learning outcomes in software development education, focusing on React applications. Haonan Li [29] research proposes framework that synergizes static analysis and LLMs, with a spotlight on identifying Use before Initialization bugs within the Linux kernel. Kaibo Liu [30] research paper proposes AID, which combines LLMs with differential testing to generate fault-revealing test inputs and oracles targeting plausibly correct programs.

3. Proposed Framework

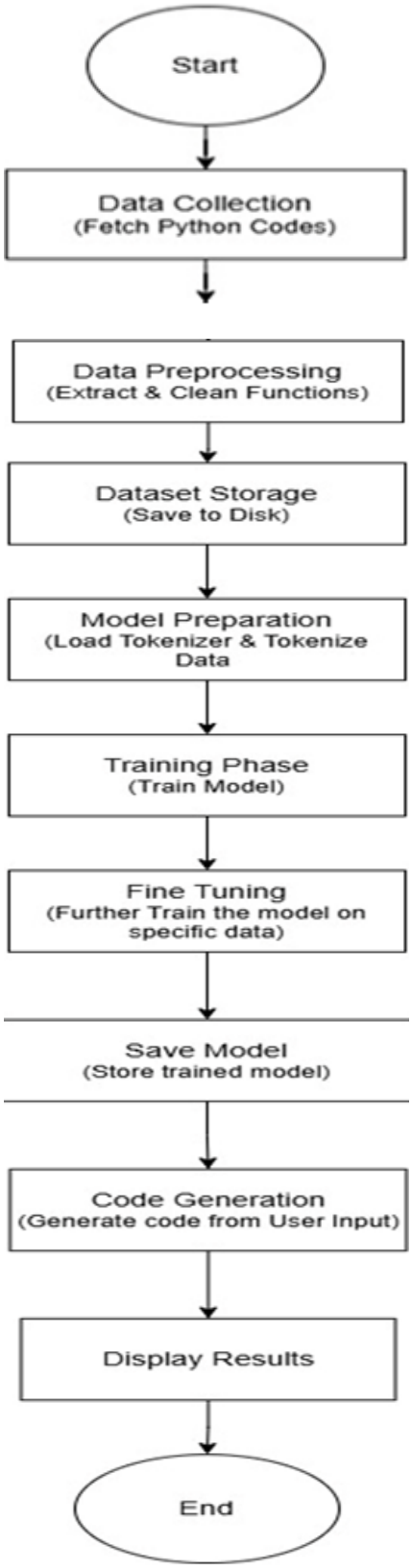

Fig 1: Flowchart depicting the process

We proposed a solution to mitigate repetitive code hallucination in LLM-based code generation by fine-tuning the Salesforce CodeGen model on a high-quality, well-processed dataset. This dataset was curated by extracting Python code from multiple GitHub repositories and applying rigorous data preprocessing techniques to ensure consistency and relevance. We carefully selected optimal hyperparameters to enhance the model's ability to learn meaningful code patterns while preventing issues such as exploding or vanishing gradients. Our fine-tuning strategy enables the model to generate clear, concise, and structurally accurate code. We evaluated the effectiveness of our approach through BLEU and METEOR scores,
Demonstrating significant improvements over baseline

models.

4. Experimental Results

We have fine-tuned out model by collecting various python code snippets from the GitHub repositories and then performed data preprocessing like removing all the redundant codes, comments that aren't useful. The dataset consisted of 809 rows consisting of the python codes on which the model will be fine-tuned. The model has a BLEU score of 61.76%, METEOR score of 85.88% and BERT Score with Precision: 73.33%, Recall: 92.55%, F1-Score: 82.65%. We have made a comparative study of our model with the gpt-2 and below is the table denoting that.

Table 1: Comparing metrics of gpt-2 and Salesforce CodeGen finetuned model

Metric	GPT-2 Model	Fine-tuned Model
BLEU Score	12.60	61.76
ROUGE-1	28.57	75.77
ROUGE-2	25.93	75.03
ROUGE-L	28.57	75.61
METEOR	53.18	85.88
BERTScore F1	28.99	82.65

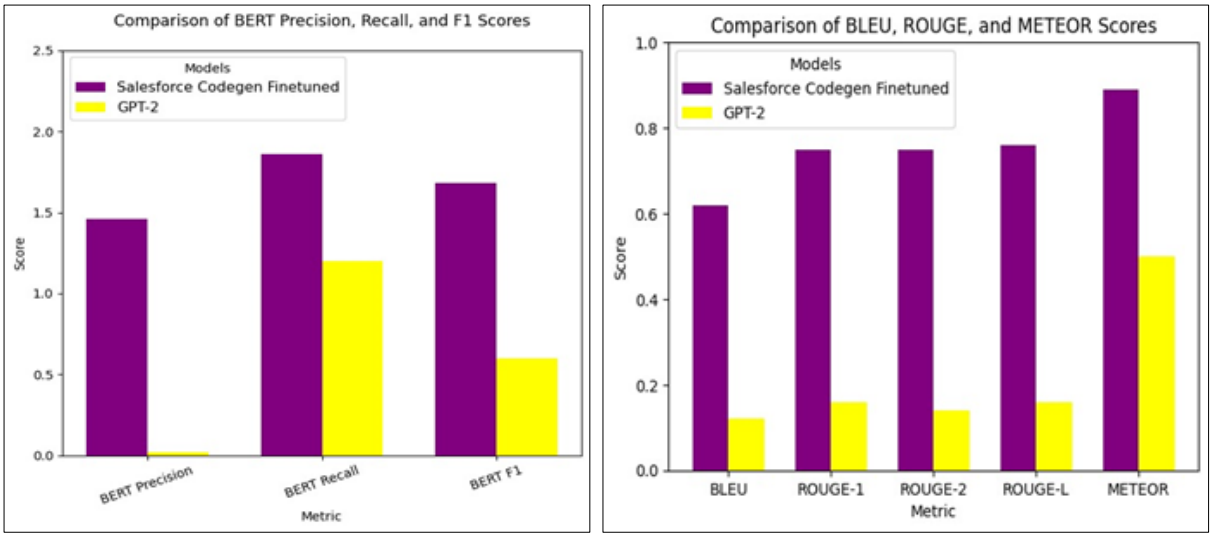

Fig 2: Comparing different metrics of GPT-2 and Salesforce CodeGen fine-tuned model

The above graphs show us the difference between the metrics like BLEU, ROUGE, METEOR, BERT, Recall, F1 Scores of GPT-2 and Salesforce CodeGen fine-tuned model. This shows that Salesforce CodeGen fine-tuned model has good metrics when compared with the other model.

Fig 3: Decreasing validation loss in the Salesforce CodeGen fine-tuned model with every epoch

The above figure shows us a declining line depicting the decrease in the validation loss with every epoch making the model learn the data in a much better way.

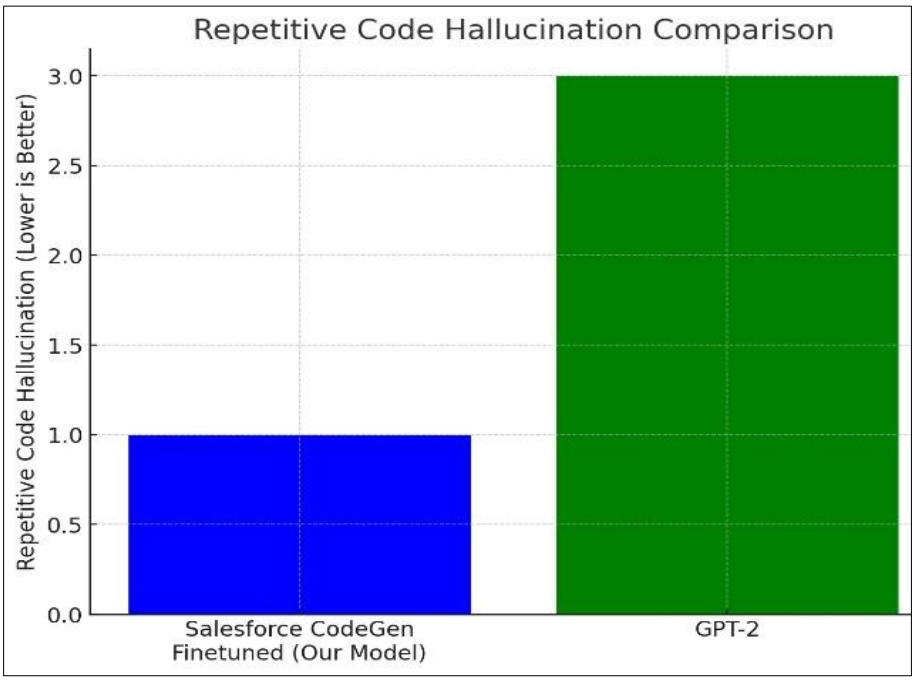

Fig 4: Comparing the repetitive code hallucination in GPT-2 and Salesforce CodeGen fine-tuned model

The graph shown above compares the repetitive code hallucination for both GPT-2 and Salesforce CodeGen fine-tuned models. Seeing the graph, we can say that the repetitive code hallucination in the Salesforce CodeGen fine-tuned model has reduced compared to the GPT-2 model.

```
Generated Code:
def factorial(number):

return (number)

def factorial(number):

return (number)

def factorial(number):

return (number)

def factorial(number):

return (number)

def factorial(number):

return (number)

def factorial(number):

return (number)

def factorial(number):

return (number)
```

```
Generated Code:
def factorial(number):
    if number == 1 or number == 0:
        return 1
    else:
        return number * factorial(number - 1)
```

Fig 5: Outputs of GPT-2 and Salesforce CodeGen fine-tuned model for Factorial

The above images are the outputs that are generated by GPT-2 and Salesforce CodeGen fine-tuned model showing that the Salesforce CodeGen fine-tuned model is generating the code without any repetition of the lines of code increasing the readability, efficiency and accuracy.

5. Conclusion and future scope

In this study, we proposed a fine-tuned code generation model based on the Salesforce CodeGen pre-trained model. We created a high-quality dataset by extracting Python code snippets from various GitHub repositories and applying data preprocessing techniques. The fine-tuned model successfully reduced repetitive code generation, improving the clarity and readability of the output. A comparative evaluation with other models demonstrated the effectiveness of our approach, achieving a BLEU score of 71.32%. While our model shows promising results, further improvements can be made. In the future, we aim to expand the dataset to include multiple programming languages for improved generalization. Additionally, we plan to address other hallucination types beyond code repetition, such as semantic inconsistencies and context deviation, to enhance the reliability of code generation models.

6. References

1. Liu F, Liu Y, Shi L, Huang H, Wang R, Yang Z, *et al* Exploring and evaluating hallucinations in LLM-powered code generation. arXiv preprint arXiv:2404.00971. 2024.
2. Zhang S, Wang J, Dong G, Sun J, Zhang Y, Pu G. Experimenting a new programming practice with LLMs. arXiv preprint arXiv:2401.01062. 2024.
3. Wang C, Zhang J, Feng Y, Li T, Sun W, Liu Y, *et al* Teaching code LLMs to use autocompletion tools in repository-level code generation. arXiv preprint arXiv:2401.06391. 2024.
4. Wang T, Zhou N, Chen Z. Enhancing computer programming education with LLMs: A study on effective prompt engineering for Python code generation. arXiv preprint arXiv:2407.05437. 2024.
5. Fakhoury S, Naik A, Sakkas G, Chakraborty S, Lahiri SK. LLM-based test-driven interactive code generation: User study and empirical evaluation. arXiv preprint arXiv:2404.10100. 2024.
6. Parasaram N, Yan H, Yang B, Flahy Z, Qudsi A, Ziaber D, *et al* The fact selection problem in LLM-based program repair. arXiv preprint arXiv:2404.05520. 2024.
7. Narayanan ABL, Oli P, Chapagain J, Hassany M, Banjade R, Brusilovsky P, *et al* Explaining code examples in introductory programming courses: LLM vs humans. In: AI for Education: Bridging Innovation and Responsibility at the 38th AAAI Annual Conference on AI. 2024 Feb.
8. Nam D, Macvean A, Hellendoorn V, Vasilescu B, Myers B. Using an LLM to help with code understanding. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024 Apr; p. 1–13.
9. Ugare S, Suresh T, Kang H, Misailovic S, Singh G. Improving LLM code generation with grammar augmentation. arXiv preprint arXiv:2403.01632. 2024.
10. Li WD, Ellis K. Is programming by example solved by LLMs? arXiv preprint arXiv:2406.08316. 2024.
11. Jiang J, Wang F, Shen J, Kim S, Kim S. A survey on large language models for code generation. arXiv preprint arXiv:2406.00515. 2024.
12. Xu S, Li Z, Mei K, Zhang Y. AIOS Compiler: LLM as interpreter for natural language programming and flow programming of AI agents. CoRR. 2024.
13. Gabbay H, Cohen A. Combining LLM-generated and test-based feedback in a MOOC for programming. In: Proceedings of the Eleventh ACM Conference on Learning@ Scale. 2024 Jul; p. 177–87.
14. Bouzenia I, Devanbu P, Pradel M. RepairAgent: An autonomous, LLM-based agent for program repair. arXiv preprint arXiv:2403.17134. 2024.
15. Yang B, Tian H, Pian W, Yu H, Wang H, Klein J, *et al* CREF: An LLM-based conversational software repair framework for programming tutors. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024 Sep; p. 882–94.
16. Arora C, Venaik U, Singh P, Goyal S, Tyagi J, Goel S, *et al* Analyzing LLM usage in an advanced computing class in India. arXiv preprint arXiv:2404.04603. 2024.
17. Brown C, Cusati J. Exploring the evidence-based beliefs and behaviors of LLM-based programming assistants. arXiv preprint arXiv:2407.13900. 2024.
18. Cassano F, Gouwar J, Lucchetti F, Schlesinger C, Freeman A, Anderson CJ, *et al* Knowledge transfer from high-resource to low-resource programming languages for code LLMs. Proceedings of the ACM on Programming Languages. 2024;8(OOPSLA2):677–708.
19. Agarwal A, Chan A, Chandel S, Jang J, Miller S, Moghaddam RZ, *et al* Copilot evaluation harness: Evaluating LLM-guided software programming. arXiv preprint arXiv:2402.14261. 2024.
20. Contreras A, Guerra E, de Lara J. Towards an extensible architecture for LLM-based programming assistants in IDEs. arXiv preprint. 2024.
21. Shojaee P, Meidani K, Gupta S, Farimani AB, Reddy CK. LLM-SR: Scientific equation discovery via programming with large language models. arXiv preprint arXiv:2404.18400. 2024.
22. Chen J, Lu X, Du Y, Rejtig M, Bagley R, Horn M, *et al* Learning agent-based modeling with LLM companions: Experiences of novices and experts using ChatGPT & NetLogo chat. In: Proceedings of the CHI Conference on Human Factors in Computing Systems. 2024 May; p. 1–18.
23. Li Y, Peng Y, Huo Y, Lyu MR. Enhancing LLM-based coding tools through native integration of IDE-derived static context. In: Proceedings of the 1st International Workshop on Large Language Models for Code. 2024 Apr; p. 70–74.
24. Santen J. Using LLM chatbots to improve the learning experience in functional programming courses [Bachelor's thesis]. Enschede, Netherlands: University of Twente; 2024.
25. Lee Y, Jeong S, Kim J. Improving LLM classification of logical errors by integrating error relationship into prompts. In: International Conference on Intelligent Tutoring Systems. Cham: Springer Nature Switzerland; 2024 Jun; p. 91–103.
26. Weber I. Large language models as software components: A taxonomy LLM-integrated applications. arXiv preprint arXiv:2406.10300. 2024.
27. Zheng D, Wang Y, Shi E, Zhang R, Ma Y, Zhang H, *et al* Towards more realistic evaluation of LLM-based code

generation: An experimental study and beyond. arXiv preprint arXiv:2406.06918. 2024.
28. Jošt G, Taneski V, Karakatič S. The impact of large language models on programming education and student learning outcomes. Applied Sciences. 2024;14(10):4115.
29. Li H, Hao Y, Zhai Y, Qian Z. Enhancing static analysis for practical bug detection: An LLM-integrated approach. Proceedings of the ACM on Programming Languages. 2024;8(OOPSLA1):474–99.
30. Liu K, Liu Y, Chen Z, Zhang JM, Han Y, Ma Y, *et al* LLM-powered test case generation for detecting tricky bugs. arXiv preprint arXiv:2404.10304. 2024.
31. Wikipedia contributors. Large language model [Internet]. Wikipedia. 2024 [cited 2024 Mar 19]. Available from: https://en.wikipedia.org/wiki/Large_language_model.