

Code Generation with Large Language Models: Auto-Completion and Bug Fixing

¹Dr. Amjad Kham, ²Kanchana S Gulannanavar

Associate Professor Department of Master of Computer Applications, BIET, Davangere-577004.

amjadkhan.biet@gmail.com, Davangere, Karnataka, India¹

P.G. Student, Department of Master of Computer Applications, BIET, Davangere-577004.

kanchanasgulannanavar@gmail.com, Davangere, Karnataka, India²

ABSTRACT

Large Language Models (LLMs) have significantly transformed software development by enabling intelligent code generation, real-time auto-completion, and automated bug fixing. Built upon transformer architectures, these models are trained on massive corpora of source code and natural language, allowing them to learn programming syntax, semantics, and structural patterns across multiple languages. This paper presents a framework for code generation using LLMs, focusing on two major applications: contextual auto-completion and automated bug repair. The proposed system integrates transformer-based architectures with fine-tuning strategies on large-scale code repositories. Experimental evaluation demonstrates improved suggestion accuracy, enhanced developer productivity, and effective bug correction compared to traditional rule-based tools. The results highlight the growing role of LLM-based assistants in modern software engineering environments.

Keywords: *Large Language Models, Code Generation, Auto-Completion, Automated Bug Fixing, Transformer Models, Software Engineering, Artificial Intelligence*

I. INTRODUCTION

Software development involves continuous writing, testing, and debugging of code. Traditional Integrated Development Environments provide limited auto-completion based on static syntax rules and local context.

These systems lack deep semantic understanding of code and therefore cannot generate complex multi-line functions or automatically repair logical errors.

The introduction of transformer architectures [1] revolutionized sequence modeling by enabling attention-based learning of long-range dependencies. Pre-trained language models such as BERT [2] and GPT [10] demonstrated strong capabilities in natural language generation and understanding. These advancements were extended to source code modeling, leading to the development of codespecific large language models such as CodeBERT [4], Codex [3], and other transformer-based models trained on massive open-source repositories.

Code generation with LLMs leverages nexttoken prediction to synthesize syntactically correct and semantically meaningful programs. Such models can generate code from natural language descriptions, provide contextual autocompletion, and even suggest fixes for buggy code. Automated program repair techniques have also evolved from rule-based systems to neural approaches that learn bug-fixing patterns from version control histories [7].

This paper presents a unified framework for auto-completion and automated bug fixing using large language models, integrating transformer-based architectures with practical deployment mechanisms in software development environments.

II. RELATED WORK

Early code generation methods relied on template-based systems and rule-driven static analysis tools. These approaches lacked adaptability and struggled with complex programming structures. Neural machine translation models introduced encoder-decoder architectures capable of learning sequence transformations [9], which were later adapted for code-to-code and text-to-code translation tasks.

Transformer-based models significantly improved representation learning through attention mechanisms [1]. BERT [2] introduced bidirectional pre-training for contextual understanding, while GPT demonstrated strong generative capabilities [10]. These developments led to domain-specific adaptations such as CodeBERT [4], which combines natural language and programming language representations.

Chen et al. [3] evaluated large language models trained on source code and demonstrated strong performance in code synthesis and completion tasks. Neural approaches to automated bug fixing were explored by Tufano et al. [7], who showed that sequence-to-sequence models could learn patterns of bug corrections from historical commits.

Graph-based program representations further enhanced code modeling by capturing structural dependencies in abstract syntax trees and control flow graphs [8]. These representations improved semantic understanding beyond simple token sequences.

Recent studies emphasize integrating LLM-based assistants into development tools, enabling real-time code completion, documentation generation, and patch suggestions. Despite their effectiveness, challenges remain in ensuring correctness, security, and explainability of generated code.

III. METHODOLOGY

The proposed framework consists of data collection, preprocessing, model training, finetuning, and deployment.

The dataset comprises large-scale open-source repositories collected from platforms such as GitHub. The dataset includes multiple programming languages and commit histories. Bug-fixing commits are extracted to construct paired datasets of buggy and corrected code snippets.

Preprocessing involves tokenizing source code into subword tokens using byte-pair encoding or language-specific tokenizers. Structural information such as indentation and brackets is preserved through special tokens. For bug repair tasks, input-output pairs are created where the input represents faulty code and the output represents the corrected version.

The model architecture is based on the transformer framework [1]. For autocompletion, a decoder-only model predicts the next sequence of tokens based on context. The model is pre-trained using next-token prediction on large-scale code datasets.

For bug fixing, a sequence-to-sequence transformer architecture is used. The encoder processes buggy code, and the decoder generates corrected code. Fine-tuning is performed using labeled bug-fix datasets to adapt the model for repair tasks.

The workflow of the system is described as follows:

Large-Scale Code Dataset Collection
Code Tokenization and Preprocessing
Pre-Training on Code Corpus
Fine-Tuning for Auto-Completion
Fine-Tuning for Bug Repair
Integration with IDE
Real-Time Suggestion and Error Correction

The model is optimized using cross-entropy loss. Evaluation metrics include top-k accuracy for auto-completion, perplexity for language modeling, BLEU score for generated code quality, and exact match accuracy for bug repair.

IV. RESULTS AND DISCUSSION

Experimental results show that transformer-based LLMs significantly outperform traditional auto-completion tools in multi-line code prediction. The models capture long-range dependencies and generate context-aware suggestions aligned with project coding styles. In bug repair experiments, the fine-tuned sequence-to-sequence model successfully corrects syntax errors and certain logical inconsistencies. Performance evaluation indicates higher repair accuracy compared to earlier neural program repair systems [7].

However, limitations include occasional hallucination of incorrect APIs, security vulnerabilities, and dependency on high-quality training data. Therefore, combining LLM outputs with static analysis tools and human validation is recommended.



Figure 4.1.2: Bug Fixing in the Python Code



Figure 4.1.1: Libraries Used

V. CONCLUSION AND FUTURE WORK

This paper presented a framework for code generation using large language models for auto-completion and automated bug fixing. Transformer-based architectures demonstrate strong capabilities in understanding programming patterns and generating syntactically valid and context-aware code.

Future work includes integrating reinforcement learning from human feedback to improve reliability, combining LLM outputs with static code analyzers for security validation, optimizing models for edge deployment, and enhancing explainability of generated solutions.

VI. REFERENCES

- [1] A. Vaswani et al., "Attention Is All You Need," Advances in Neural Information Processing Systems, 2017.
- [2] J. Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," NAACL, 2019.
- [3] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv, 2021.
- [4] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," EMNLP, 2020.
- [5] Y. Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach," arXiv, 2019.
- [6] T. Brown et al., "Language Models are FewShot Learners," NeurIPS, 2020.
- [7] R. Tufano et al., "An Empirical Study on Learning Bug-Fixing Patches in the Wild," IEEE Transactions on Software Engineering, 2019.
- [8] M. Allamanis et al., "Learning to Represent Programs with Graphs," ICLR, 2018.
- [9] D. Bahdanau et al., "Neural Machine Translation by Jointly Learning to Align and Translate," ICLR, 2015.
- [10] T. Brown et al., "Language Models are Few-Shot Learners," NeurIPS, 2020.
- [11] J. Weston et al., "Learning to Execute," ICLR, 2016.
- [12] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," ICLR, 2015.