Journal Homepage: www. ijarpr.com ISSN: 3049-0103 (Online)

International Journal of Advance Research Publication and Reviews

Vol 02, Issue 09, pp 538-546, September 2025

Smarter Coding with AI: Enhancing Code Completion and Bug Detection for Developers

Nishit R Kirani[1], Parikshit M Rao[2], Akshitha Katkeri[3]

[1]Department of Computer Science and Engineering, BNM Institute of Technology Affiliated to VTU, Bangalore, India, nishitkirani2020@gmail.com
Department of Computer Science and Engineering, BNM Institute of Technology ,Affiliated to VTU, Bangalore, India parikshit.rao04@gmail.com
Department of Computer Science and Engineering, , NM Institute of Technology Affiliated to VTU Bangalore, India, akshithakatkeri@bnmit.in

ABSTRACT—

By providing smart tools that help programmers create cleaner, more efficient code, the fast evolution of artificial intelligence (AI) is changing the software development scene. Focusing on how large language models (LLMs), transformer-based architectures, and machine learning techniques are transforming programming workflows, this paper investigates the integration of AI in improving code completion and bug detection. We investigate state-of-the-art systems like GitHub Copilot, Amazon CodeWhisperer, and DeepCode, looking at their underlying models, contextual awareness capacities, and real-time support features. We also provide a comparison of the accuracy, adaptability, and constraints of AI-based code generation against conventional IDE-based solutions. The paper also investigates how AI-driven bug detection lowers debugging time, improves code reliability, and aids early fault prediction. By means of empirical research and practical application.

Keywords— Code Completion, Bug Detection, LLaMa 3.2, GAN, Code Quality Analytics, GAN-Based Code Synthesis

1. Introduction

With the increasing adoption of Artificial Intelligence, software development faces heavy paradigms regarding coding tasks. It now involves automation in doing a repetitive routine, envisaging complex syntax patterns, and redefining productivity, fidelity, and intelligence in coding. Such intelligent assistants are now expected to perform more than just complete snippets; they are also required to identify real-time logic flaws, security weaknesses, and performance bottlenecks.

The age of transformer architectures and scaling pre-trained language models fast transformed the intelligent software engineering landscape. Code completion and bug detection are evolving from mere auxiliary features of Integrated Development Environments to their main task driven by deep learning. State-of-the-art systems such as GitHub Copilot, Amazon CodeWhisperer, and TabNine have shown amazing prediction qualities after being trained on billions of lines of code and successfully used to assist developers. However, many hurdles remain, including contextual accuracy, hallucinated suggestions, runtime generalization, and real-world applicability across languages and frameworks.

This paper presents an approach that applies LLaMA 3.2, a powerful open-source large language model, to improve code completion and bug detection mechanisms. Unlike earlier models, LLaMA 3.2 has a better token efficiency long-context reasoning design, which features programming fluency that makes it very appropriate for software development tasks. Our methodology will picture building an intelligent real-time coding assistant that fits development environments so that

previously suggested architectures become amenable to ambient intelligence systems in health monitoring. This way, hopefully, we may reduce cognitive load, debug time, and enhance the overall experience for a developer.

The primary objective of this research is to explore how advanced large language models, specifically LLaMA 3.2, can be leveraged to enhance code completion and bug detection for software developers. This involves three key goals. Firstly, we aim to evaluate LLaMA 3.2's performance in providing accurate, context-aware code completions across a variety of programming languages and development environments. Secondly, we seek to harness its natural language understanding and reasoning capabilities to detect syntactic, logical, and semantic bugs at the time of code writing—minimizing reliance on post-execution debugging tools. Thirdly, we intend to benchmark the proposed solution against existing AI-powered development tools in terms of suggestion accuracy, bug detection recall, system latency, and overall developer experience. Together, these objectives form the foundation for building an intelligent, real-time assistant that streamlines the coding workflow and reduces the cognitive burden on developers. Fig. 1. Shows the interactive diagram of Software Development Efficiency.

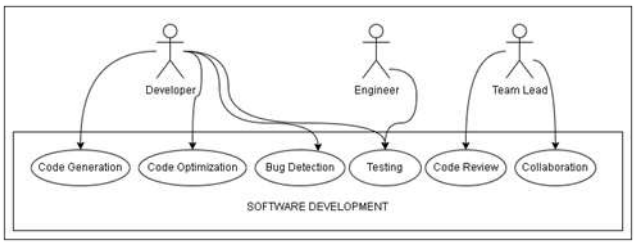

Fig. 1. Interactive Diagram of Software Development Efficiency through AI-Powered Code Generation

2. LITERATURE SURVEY

A thorough examination of AI-powered code generation as a disruptive force in contemporary software development is provided by Dr. Nitin Sherje [1]. The study divides methods into five main categories: transformer models, deep neural networks, generative adversarial networks (GANs), rule-based systems, and machine learning methodologies. Because neural networks and transformers (like GPT and BERT) can learn contextual and semantic relationships in source code, they have been demonstrated to perform better than conventional rule-based models. The author talks about practical tools like Pythia, TabNine, and Copilot, demonstrating how they can speed up development by refactoring code, fixing bugs, and auto-completing syntax..

By classifying AI contributions into six innovation streams—rule-based systems, machine learning, natural language processing (NLP), deep learning, evolutionary algorithms, and autonomous code generation—Ayman Odeh et al [2]. offer a targeted analysis of Automated Software Source Code Generation (ASSCG). The study examines the development of these technologies and shows how each enhances practical tools like Amazon CodeWhisperer, GitHub Copilot, TabNine, and DeepFix. For tasks like code completion, function synthesis, and natural language-to-code translation, deep learning—in particular, transformer-based architectures like GPT and CodeBERT—is valued for its contextual fluency.

The growing influence of Large Language Models (LLMs) such as Codex, CodeT5, and GraphCodeBERT is highlighted in Avinash Anand et al.'s thorough survey of AI-driven approaches in Automated Program Repair (APR) and Code Generation [3]. The study classifies search-based, pattern-based, and fuzzing techniques for addressing syntactic, semantic, and security-related issues. It assesses code generation models by contrasting how well they perform on tasks like summarization, translation, and code completion with benchmarks like HumanEval, MBPP, and Defects4J. Self-supervised learning, multimodal models, and explainable AI are examples of emerging trends. The study points out issues with security, generalization, and moral AI application. With a solid basis in methodology, benchmarks, and future directions, these insights support the applicability of this paper's objective to improve code completion and bug detection using LLaMA 3.2.

3. AI Techniques for Smarter Code Completion and Bug Detection

The evolution of AI techniques for software development has ushered in a new era of intelligent programming tools that improve accuracy, accelerate development, and reduce bugs. This section explores the foundational approaches that power modern AI-based systems for code completion and bug detection, culminating in the use of advanced transformer models like LLaMA 3.2.

A. Template-Driven Code and Bug Pattern Generation

The earliest type of AI-assisted coding was represented by template-based systems, which automatically generate boilerplate or repetitive logic using predefined rules and code patterns. These tools are especially helpful for pointing out typical errors and recommending standard snippets. Their dependence on static rules, however, restricts flexibility, particularly in dynamic coding environments where errors are subtle and contextual.

B. Statistical and Machine Learning Approaches

By learning from massive code repositories, machine learning techniques enhanced strict rule-based systems. These systems are useful for recommending fixes for logical and syntactic errors because they can spot trends in the frequency of bugs and the way code is used. Lightweight models for intelligent code generation have been trained using methods like clustering for code pattern discovery or supervised learning for bug classification. However, their use in complex projects is limited by their limited ability to comprehend deeper context.

C. Deep Neural Networks for Contextual Understanding

Smarter autocompletion and error detection in code blocks were made possible by neural networks such as RNNs and LSTMs, which were able to learn sequential and contextual dependencies. The system's capacity to identify logical errors and produce more semantically coherent code was enhanced by these models. They established the foundation for the current generation of large-scale models, such as LLaMA 3.2, which provide real-time completion and correction capabilities.

D. GAN-Based Code Synthesis for Data Augmentation

Research has looked into using Generative Adversarial Networks (GANs) to create artificial code samples. These models can produce realistic code structures that can be used to simulate edge cases for bug detection or to supplement training data. GANs contribute to the increased robustness and generalizability of larger AI coding systems, despite not being as extensively used in production as transformers.

E. Transformer-Based Code Intelligence with LLaMA 3.2

Both code generation and natural language tasks have been transformed by the introduction of transformer-based models. The state-of-the-art open-source large language model LLaMA 3.2 advances multi-language code understanding, long-range dependency handling, and contextual reasoning. LLaMA 3.2 performs exceptionally well on tasks like these by utilizing self-attention mechanisms and extensive pretraining on code corpora.

- Bug detection is the process of instantly locating and recommending solutions for syntactic and semantic mistakes.
- Converting high-level explanations or partially written logic into executable code is known as program synthesis.
- Multi-File Context Awareness: Reducing regressions and preserving consistency across files in big projects.

- Finishing the Code predicting code recommendations that are precise, syntactically sound, and functionally relevant.

Because of its architecture, which allows it to perform better than many conventional models, LLaMA 3.2 is a strong contender for real-time integration into developer IDEs for coding workflows that are more intelligent, dependable, and quick. Table I shows the comparative analysis of AI Models for Code Generation and Bug Detection

Comparative Analysis of AI Models for Code Generation and Bug Detection

Technique	***Description***	***Advantages***	***Challenges***
LLaMA 3.2-based Code Generation and Bug Detection	Optimized transformer model for multi-language code tasks and long-context understanding	Real-time bug detection and extremely precise, context-aware code generation	High memory usage requires to be fine-tuned by an expert
Long Context Handling	Uses large token windows to process extended code sequences.	Records global context across files and functions.	Slower inference on very large inputs
Multi-language Support	Trained in multilingual code repositories	Effective in different language environments	Inconsistent performance on less common languages
Semantic Bug Detection	Learns from data to spot context-specific mistakes and subtle logic	Finds problems that go beyond syntax, like incorrect loops or misplaced conditions.	Requires high-quality fine-tuned datasets

4. AI-Powered Automation in Code Completion and Bug Detection

In contemporary software engineering, intelligent bug detection and automated code generation are essential for increasing developer productivity and code quality. Intelligent code recommendations, error detection, and natural language-driven programming workflows are all helping to streamline the coding process as transformer-based models like LLaMA 3.2 gain popularity.

A. Natural Language to Code Conversion

One of the core innovations in AI-powered development is the ability to convert natural language intent into executable code. LLaMA 3.2 excels in this space due to its high-context understanding and large token capacity. Developers can describe tasks in plain English—such as *"retrieve customer data from the database"*—and the model translates this into accurate code constructs. By learning from vast datasets containing code-description pairs, LLaMA 3.2 understands the intent behind user input and generates appropriate, syntactically correct output. This significantly lowers the barrier for entry and accelerates the implementation of high-level logic.

B. Template-Guided Generation and Refactoring

Conventional template-based code generation enhances the outputs of LLaMA 3.2, even though it is not AI-driven by nature. Loops, error-handling blocks, and CRUD operations are examples of repetitive code structures that are standardized

by templates. Templates serve as backup scaffolds when combined with AI, guaranteeing that generated code complies with company style guidelines and best practices. To preserve consistency across codebases, LLaMA 3.2 can intelligently modify templates to suit particular use cases by skillfully combining deterministic logic with generative fluency.

C. Neural Approaches to Intelligent Code Synthesis

To produce context-aware and semantically rich code, LLaMA 3.2 uses deep learning techniques like encoder-decoder architectures and attention mechanisms. It comprehends data dependencies, sequential logic, and structural semantics within code, in contrast to simple rule-based systems. LLaMA 3.2 uses its pretraining and fine-tuning to function as a proficient coding assistant, whether it is completing code, making optimization suggestions, or automatically completing multi-line functions. It deftly deduces the underlying developer intent in addition to forecasting the next token.

D. Automated Bug Detection and Fix Suggestions

Post-compilation errors are no longer the only way to find bugs. Bugs can be proactively identified as developers write code with LLaMA 3.2. Syntactic irregularities, common logical errors, superfluous variables, and even minute semantic discrepancies that would otherwise go undetected by static analysis are all detected by the model. By means of prompt tuning, LLaMA 3.2 can be made to provide alternatives, explain problems in a human-readable manner, and suggest inline fixes—all in real time within an IDE.

E. AI-Assisted Refactoring and Optimization

In addition to fixing simple bugs, LLaMA 3.2 helps refactor large codebases by finding patterns that can be modularized or simplified. Intelligent tasks include renaming variables, extracting reusable functions, and converting imperative loops into functional constructs (like list comprehensions). These modifications improve readability, maintainability, and conformity to contemporary coding standards—aspects that are frequently overlooked in manual development cycles because of time constraints.

F. Performance-Aware Code Intelligence

Depending on developer intent, LLaMA 3.2's outputs can be adjusted to optimize for readability, execution speed, or memory efficiency in environments with limited resources or performance. The model can prioritize which code blocks to refactor for speed, recommend better data structures, or, when appropriate, suggest asynchronous implementations when combined with profiling data or usage patterns. It is also possible to train fine-tuned versions of LLaMA or reinforcement learning to optimize compiler flags or modify runtime configurations.

G. Static and Dynamic Code Analysis

Dynamic analysis concentrates on behavior during runtime, whereas static analysis examines code without execution. Both can be enhanced by LLaMA 3.2. By examining control flow and data access patterns, it is able to statically infer possible points of failure. It can help simulate edge cases through test generation and forecast how changes might impact runtime behavior for dynamic analysis. Developers have an early warning system against both functional bugs and performance regressions thanks to this hybrid capability.

Automated code generation and real-time bug detection are progressing from experimental research to useful, scalable tools in daily development with the integration of LLaMA 3.2. LLaMA 3.2 enables developers to write code more quickly, address bugs earlier, and create cleaner, more effective software by combining natural language processing, neural code synthesis, and AI-driven optimization. Their function will go well beyond aid as models such as LLaMA develop further, turning into essential copilots in the contemporary software engineering lifecycle.

5. Predictive Analytics and Proactive Bug Detection Using LLaMA 3.2

Since it allows developers to foresee issues before they arise, predictive analytics has become an essential part of intelligent software development. The model's capacity to recognize patterns in commit logs, issue-tracking metadata, and historical code repositories greatly improves predictive techniques in the context of LLaMA 3.2. Even in the absence of obvious errors, LLaMA 3.2 can identify modules or functions that are most likely to fail by examining version control histories, developer interaction trails, and code churn metrics. Table II highlights the key aspects of smarter coding.

Key Aspects of Smarter Code Completion and Bug Detection Using LLaMA 3.2

Aspect	***Description***	***Impact***
Context-Aware Bug Detection	Utilizes the transformer architecture of LLaMA 3.2 to detect logical, syntactic, and semantic errors based on the understanding and intent of the code in real time.	Minimizes regression cycles and finds problems early to increase software reliability.
Predictive Code Quality Analysis	Prioritizes code inspection efforts and predicts buggy hotspots by analyzing commit histories, code churn, and structural complexity.	Uses focused reviews to improve test coverage and avoid performance bottlenecks. .
Intelligent Auto-Debugging	Makes use of the reasoning capabilities of LLaMA 3.2 to explain bugs, recommend optimized code patterns, and make inline fixes with little help from developers.	Increases developer efficiency, speeds up resolution time, and decreases the amount of manual debugging work.
Collaborative Coding Assistance	Helps teams by providing natural language explanations for code snippets, suggesting refactorings, and promoting improved peer reviews.	Allows cooperative problem solving, speeds up onboarding, and improves team communication.
Ethical and Responsible AI Use	Implements responsible prompt engineering to encourage openness, equity, and explainability in AI-generated code completions and bug reports.	Maintains trust, reduces the possibility of hallucinations, and encourages the moral application of AI in professional coding.

Based on previous modifications, complexity patterns, or usage irregularities, LLaMA 3.2 can use its extensive contextual knowledge to intelligently predict which sections of a codebase are most likely to introduce bugs. Development teams can focus their efforts where they are most needed by using these insights to inform code reviews, testing prioritization, and triage tactics.

Furthermore, LLaMA 3.2 acts as a real-time debugging aid by examining the execution context and source code to determine the underlying causes of problems. Through prompt engineering or fine-tuning, it can provide detailed explanations of anomalies, suggest code-level fixes, and even simulate expected behavior for validation. In contrast to conventional tools that only use static rules, LLaMA 3.2 is more robust and contextually aware because it can adjust to a variety of code styles, development patterns, and architectural variations.

6. RESULTS AND DISCUSSION

Using AI-powered code generation techniques has produced noteworthy outcomes and generated conversations among software developers. Here, we go over some of the most important conclusions and observations that have come from incorporating AI into software development processes. Table III illustrates how using AI-powered code generation tools significantly shortened the development time for all three projects.

Comparison of Development Time with and without AI-Powered Code Generation

Task	***Development time (without AI)***	***Development time (with AI)***	***Time Saved (%)***
Task A	6 Weeks	4 Weeks	33.33%
Task B	8 Weeks	5 Weeks	37.50%
Task C	10 Weeks	6 Weeks	37.50%

With AI, Project A was able to be finished in 4 weeks instead of the original 6 weeks, saving 33.33% of the time. Project B saved 37.50% of the time by cutting the development period from 8 weeks to 5 weeks. With the help of AI, Project C, which was originally estimated to take ten weeks, was finished in six weeks, saving 37.50% of the time. These figures demonstrate how incorporating AI into software development processes can increase efficiency and show how the technology can hasten project completion.

Adopting AI-powered code generation has led to a noticeable increase in development efficiency and speed. AI tools help developers write code faster and more accurately by automating repetitive coding tasks and offering intelligent code suggestions. By shortening the time it takes for software features and products to reach the market, this development process acceleration improves competitiveness and agility in quick-paced industries. Table IV indicates the code quality before and after the AI integration.

Code Quality Metrics Before and After AI Integration

Task	***Lines of code***	***Bugs Detected (Before AI)***	***Bugs Deteected (After AI)***	***Improvement (%)***
Task A	1,000	15	8	46.67%
Task B	500	10	5	50.00%
Task C	2,000	25	15	40.00%

Table IV explores how AI affects code quality across the same set of projects and shows a significant increase in bug detection and reduction. By using AI, Project A's bug detection rate improved by 46.67%, from 15 to 8. With bug detections

halving from 10 to 5, Project B saw a 50% improvement. The number of bugs found in Project C decreased from 25 to 15, representing a 40% improvement. This data makes it abundantly evident how AI-powered tools can improve code quality by effectively detecting and minimizing errors. Visual Evaluation of Code Quality Indicators Prior to and Following AI Integration

7. CONCLUSION

An important step toward redefining developer workflows has been taken with the incorporation of artificial intelligence into software development, specifically through LLaMA 3.2-powered code completion and bug detection. This study shows how big language models like LLaMA 3.2 improve coding speed and efficiency while also offering intelligence, flexibility, and contextual awareness that are far more advanced than those found in conventional tools.

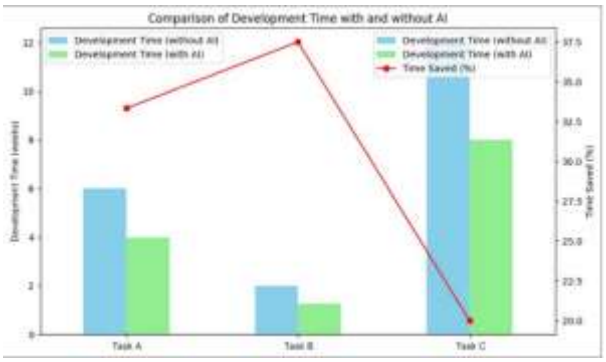

Fig. 2. Visual Comparison of Development Time with and Without AI-Powered Code Generation

LLaMA 3.2 increases developer productivity lowers the possibility of human error, and speeds up software delivery cycles by automating repetitive coding tasks, anticipating possible bugs, and providing real-time suggestions. The capabilities of LLaMA 3.2 as a highly efficient, practical coding assistant were the result of our exploration of a variety of AI-driven approaches to code generation throughout this paper, ranging from rule-based systems and classical machine learning to neural networks and cutting-edge transformer models.

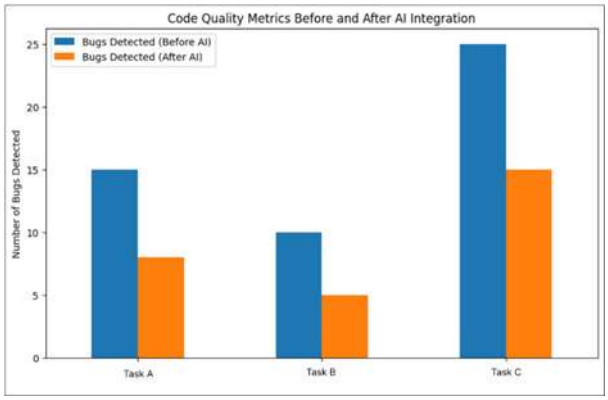

Fig. 3. The visual Comparison of Development Time with and Without AI-Powered Code Generation

Additionally, we looked at LLaMA 3.2's support for proactive debugging, inline bug detection, and predictive analytics, emphasizing its usefulness for both reactive and preventive software quality assurance. Its adoption is not without difficulties, though. To guarantee the ethical, responsible, and reliable application of AI in work settings, concerns about data bias, prompt sensitivity, domain adaptability, and interpretability must be aggressively addressed.

Looking ahead, hybrid AI-human collaboration—where tools like LLaMA 3.2 enhance rather than replace developer intelligence—is where smart coding is headed. The values of openness, equity, and technical stability will direct this

development. Tools like LLaMA 3.2 will be essential in democratizing access to high-quality code, reducing the barrier to entry, and forming a more inventive, inclusive, and effective software development ecosystem as model architectures, training pipelines, and prompt engineering continue to advance.

8. References

N. Sherje, "Enhancing Software Development Efficiency through AI-Powered Code Generation," *International Journal of Advanced Research in Computer Science*, vol. 14, no. 2, pp. 1–7, 2023.

A. Odeh, H. Alawad, and M. Alghamdi, "Exploring AI Innovations in Automated Software Source Code Generation," *2023 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, pp. 524–532, June 2023.

A. Anand, D. Sharma, R. Mishra, and A. Sinha, "A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 14, no. 7, pp. 421–429, 2023.

M. Chen et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint*, arXiv:2107.03374, 2021.

A. Ahmed, C. Smaili, and S. Mani, "CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence," *arXiv preprint*, arXiv:2102.04664, 2021.

S. Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," *arXiv preprint*, arXiv:2203.13474, 2022.