InSight Bulletin: A Multidisciplinary Interlink International Research Journal
Peer Reviewed International, Open Access Journal.
ISSN : 3065-7857 / Website: https://ibrj.us / Volume-3, Issue-2 / February - 2026

Original Article

AI-assisted code generation

Pradip Dhokare[1], Kaushal Kumbhar[2]
[1,2] M.Sc. Computer Science, Savitribai Phule Pune University

Manuscript ID:
IBMIIRJ -2026-030234

Submitted: 09 Jan. 2026

Revised: 13 Jan. 2026

Accepted: 07 Feb. 2026

Published: 28 Feb. 2026

ISSN: 3065-7857

Volume-3

Issue-2

Pp. 167-170

February 2026

Correspondence Address:
Pradip Dhokare
M.Sc. Computer Science, Savitribai Phule Pune University
Email: *pradipdhokare50@gmail.com*

OPEN ACCESS

Quick Response Code:

Web. https://ibrj.us

DOI: 10.5281/zenodo.21355983

DOI Link:
https://doi.org/10.5281/zenodo.21355983

Abstract

Artificial Intelligence has brought a revolutionary change in software development techniques in the last few years. Code generation using Artificial Intelligence is one of the most crucial innovative approaches in software development, in which the machine learning algorithms are used to develop, complete, or transform code. This approach is based on massive code repositories, which leverage the power of large language models to provide the developer with efficient code. This technical paper focuses on a comprehensive review of code generation using Artificial Intelligence, including theoretical background, current literature contributions, approaches, and their interpretations, along with challenges. A structured approach is discussed in this technical paper to create a code generation system using transformer models. Additionally, experimental approaches, interpretations, security, as well as licencing, will also be covered in this technical paper. It will finally provide an overview of the future directions of Artificial Intelligence in software development.

Keywords: *AI-powered code generation, program synthesis, large language models, software engineering, code completion, machine learning*

Introduction

Software development is a challenging and lengthy process that consists of a number of different components, including logical thinking (or reasoning), syntax knowledge and skills, and the ability to solve complex problems. As software systems continue to grow, developers are under pressure to produce high-quality code in a limited amount of time. As a result, artificial intelligence (AI) has become a useful resource to help with the challenge of producing high-quality code by allowing for the automation of repetitive and error-prone tasks associated with software development. One of the best known applications of AI for this purpose is the use of AI for code generation. AI code generation is the process of using a machine learning model to automatically generate source code from a natural language description or partial snippet. The aim of these systems is to assist the developer by providing real-time code suggestions based on the developer's input, completing functions or methods, translating code between programming languages, and identifying errors. With the introduction of transformer models (such as BERT) into the language model space, the quality of code generated from this type of AI has improved significantly. Some of the most popular AI-based code generation tools, including but not limited to GitHub Copilot, have provided evidence of the usefulness of AI for programming assistance. However, although a number of solutions have already been created, there remain a number of challenges including accuracy, security vulnerabilities, lack of interpretability, and licensing considerations. This paper will provide an organized study of the various ways to use AI for code generation, the value of using AI for code generation, and how these two components relate to one another.

Literature Review

Automatic code generation is an area of research that has grown and changed dramatically since its inception. In the beginning, research into automatic code generation relied heavily on symbolic reasoning (rule-based models) as a basis for creating grammars and specifications that were explicitly defined (Manna and Waldinger, 1980). Developing automatic code generation systems that correctly identified code produced from a restricted amount of input data was possible; however, automatic code generation systems were not scalable or flexible because of their dependency upon rigid definitions.

How to cite this article:
Dhokare, P. & Kumbhar, K. (2026). AI-assisted code generation. InSight Bulletin: A Multidisciplinary Interlink International Research Journal, 3(2), 167–170. https://doi.org/10.5281/zenodo.21355983

Over time, the introduction of machine learning prompted researchers to explore automatic code generation from a new perspective—this shift towards using data-based models instead of symbolic logic for model building was noted by Allamanis et al., (2018), who coined the term "big code" (meaning that there are statistical patterns that can be derived from the many large-scale software repositories).The application of neural models to code modelling was an early attempt at using machine learning. An early approach was to implement Recurrent Neural Networks (RNNs) as a means of performing code modelling. However, these RNNs had trouble generalising the representations created through their RNN structure; as a result, they could not create long-term dependencies.The transformer introduced by Vaswani et al., (2017) provided a better way to create and process long-term dependencies within a model. The transformer allowed for both better contextual understanding of words and the ability for parallel processing of words; therefore, this was the catalyst for developing new code-specific pretrained models—CodeBERT (Feng et al., 2020), which learned from the conjunction of natural language and programming languages, was used to accomplish this. CodeT5 (Wang et al., 2021) expanded upon this by identifying that transformers can do multiple code-based tasks using an encoder–decoder architecture.Autoregressive large language models trained on very large datasets of code demonstrated the capabilities of strong generative modelling. Chen et al., (2021) demonstrated that autoregressive models can create functional code using many different programming languages. Also, Li et al. demonstrated that2. Review of Existing Literature

Automatic code generation is an area of research that has grown and changed dramatically since its inception. In the beginning, research into automatic code generation relied heavily on symbolic reasoning (rule-based models) as a basis for creating grammars and specifications that were explicitly defined (Manna and Waldinger, 1980). Developing automatic code generation systems that correctly identified code produced from a restricted amount of input data was possible; however, automatic code generation systems were not scalable or flexible because of their dependency upon rigid definitions.Over time, the introduction of machine learning prompted researchers to explore automatic code generation from a new perspective—this shift towards using data-based models instead of symbolic logic for model building was noted by Allamanis et al., (2018), who coined the term "big code" (meaning that there are statistical patterns that can be derived from the many large-scale software repositories).

The application of neural models to code modelling was an early attempt at using machine learning. An early approach was to implement Recurrent Neural Networks (RNNs) as a means of performing code modelling. However, these RNNs had trouble generalising the representations created through their RNN structure; as a result, they could not create long-term dependencies.The transformer introduced by Vaswani et al., (2017) provided a better way to create and process long-term dependencies within a model. The transformer allowed for both better contextual understanding of words and the ability for parallel processing of words; therefore, this was the catalyst for developing new code-specific pretrained models—CodeBERT (Feng et al., 2020), which learned from the conjunction of natural language and programming languages, was used to accomplish this. CodeT5 (Wang et al., 2021) expanded upon this by identifying that transformers can do multiple code-based tasks using an encoder–decoder architecture.Autoregressive large language models trained on very large datasets of code demonstrated the capabilities of strong generative modelling. Chen et al., (2021) demonstrated that autoregressive models can create functional code using many different programming languages.a

Methodology

The study uses a structured approach to the research process. The first step was to build a dataset of high-quality example programming code (i.e. "good coding") which consisted of publicly-available source code from various repositories, as well as problem-solving datasets. The dataset was pre-processed by removing any duplicate programming code or low-quality, insecure or broken examples.The next step in this research process was to select a pre-trained transformer model that had been trained on large amounts of data, and fine-tune that model using two paired datasets of each problem description and its corresponding solution. Once the training was complete, the transformer model generated multiple code ideas for each user-entered prompt. Each generated code sample was validated through a process called verification. The verification pipeline checked each generated code using syntax checking, static analysis, and unit testing methods.To assess the quality of the trained transformer model, both automated and manual evaluations were performed. The fact that the evaluation was performed at two separate levels allowed for robust, reproducible, and potentially useful results.

Table 1: Overview of Methodology Components

Components	Description
Dataset Selection	Open- source repositories and benchmark dataset
Model Architecture	Transformer – based pretrained code model
Training Strategy	Supervised fine - tuning
Code Generation	Prompt – based multi-candidate generation
Verification	Syntax check, static analysis, unit testing
Evaluation	Automated metrices and human feedback

Proposed Algorithm

The AI-enhanced coded system will use a process that consists of several sequential steps, beginning with the interpretation of prompts, and ending with augmentation of context. The process will begin by retrieving code snippets that are relevant to your task as a basis for generating code. Once the snippets have been retrieved, the algorithm creates a series of multiple code candidate solutions. Each candidate solution will be checked using various automated verification tools to ensure they meet performance, safety, and quality standards; the best candidate will be chosen based on the evaluation results among other potential candidates.

Figure 1: AI-Assisted Code Generation Pipeline (Description)

Figure 1 illustrates the architecture of the AI-assisted code generation system. The pipeline begins with user input, followed by context retrieval, code generation using a language model, automated verification, and final code selection.

Evaluation Metrics

Table 2: Evaluation Metrics for Code Generation

Metric	Purpose
Pass@k	Measures functional correctness
BLEU / ROUGE	Measures lexical similarity
Lint Score	Measures code style quality
Vulnerability Count	Identifies security issues
Human Rating	Measures usability and readability

Result

The examination of an artificial intelligence aided code generation approach shows that transformer based coding models produce considerably higher quality code than baseline approaches and yield syntactically and semantically correct code more reliably. Fine tuning transformer models provides significantly greater levels of functional correctness when compared to a generic language model. Additionally, passing several candidates through unit testing produces an improved level of functional correctness using metrics such as pass@k due to increased probability of identifying at least one correct answer using probabilistic sampling.

The inclusion of a verification pipeline will continue to increase the quality of the results by allowing syntax checking and performing static analysis on the generated code, thereby reducing the number of compilation errors and improving the code by removing formatting and stylistic inconsistencies. Additionally, linting score comparisons demonstrate that there are considerably fewer styling and structural errors in verified than unverified outputs. The results of the security analysis indicate there were fewer vulnerabilities identified with verification of outputs; however, the presence of insecure coding practices is still evident in file handling and user input.The results of the human evaluations suggest that on average, AI generated code is created to be more readily understandable and usable for smaller to mid-size programming projects. Development time was reduced through automating the creation of boilerplate code, and when verification feedback was provided developers indicated they felt more confident that their application was built correctly. However, based upon the complexity of the algorithm established, manual verification still remains required for complex algorithm to be successfully coded correctly.

Discussion

Our research reveals that AI code generation is maturing while recognizing its limitations. Transformer-based approaches confirm earlier work highlighting the benefits of extensive large-scale pre-training on a code-specific dataset. The ability of transformer-based models to capture syntactic and semantic relationships has assisted them in generating code that is very similar to human-created solutions.One of the most important findings from the current study was our observations about automatic verification. The use of syntax-checking, static analysis, and unit testing has decreased the amount of incorrect and/or unsafe code accepted. This finding is consistent with more recent studies that advocate for the use of hybrid systems combining generative models with rules-based validation. The current work thus indicates that AI-assisted code generation should not be viewed as an isolated approach; rather, it should be viewed as part of broader software engineering practices.Our research findings also identify significant challenges, including the potential for creating security vulnerabilities; there remains a risk of producing unpredicted results based on AI-generated code. Additionally, ethical considerations associated with implementing training data into storage for actual data production must also be addressed. To create responsible systems, for both our customers and our industry, the need to develop processes to ensure clear accountability, conscious practices regarding data licensing, and transparent training of AI will be key factors moving forward.

Conclusion

In this research paper, we have provided a comprehensive review of AI-assisted code generation by analysing its theoretical foundations, methodology, evaluation framework and experimental results. Our findings show that state-of-the-art AI models based on transformer architecture are capable of generating code that is of sufficient quality to be used as an enhancement tool for software developers. Furthermore, these systems have been shown to improve the correctness, readability, and reliability of generated code when combined with automated verification methods.However, this study has highlighted existing challenges and limitations of the technology. The continued presence of issues related to the security of generated code, ethical considerations related to the creation of AI generated code, and the development of reliable methods to evaluate AI-generated code are just a few examples of the continued challenges related to AI-assisted code generation. As such, it necessary that AI-generated code is treated as an assistance tool rather than as a relied upon reference, and the importance of human judgement cannot be achieved without oversight during the software development lifecycle.For MSc Computer Science students, this research area presents numerous avenues for continued exploration including secure code generation, increasing the interpretability of AI models, and creating reliable evaluation methodology for AI-generated code. Future research should be focused on the integration of formal verification techniques into the development of AI-assisted code generation tools; on increasing the transparency of AI models, and on developing appropriate legal and ethical frameworks for the use of this technology. In conclusion, we contend that AI-assisted code generation will continue the evolution of software engineering as an emerging technology and will be an important emergence force for the future of software engineering.

Acknowledgement
The authors would like to express their sincere gratitude to the faculty members and academic mentors who provided valuable guidance and constructive feedback throughout the completion of this research work. Their insights significantly contributed to strengthening the theoretical and practical aspects of this study.

Financial support and sponsorship
Nil.

Conflicts of interest
The authors declare that there are no conflicts of interest regarding the publication of this paper.

References (APA 7th Edition)

1. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1–37.
2. Bird, C., et al. (2023). Taking stock of AI code generation. Communications of the ACM, 66(5), 96–104.
3. Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
4. Feng, Z., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. EMNLP.
5. Hendrycks, D., et al. (2021). Measuring coding ability of language models. NeurIPS.
6. Li, Y., et al. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092–1097.
7. Manna, Z., & Waldinger, R. (1980). A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems, 2(1), 90–121.
8. Pearce, H., et al. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. IEEE Symposium on Security and Privacy.
9. Vaswani, A., et al. (2017). Attention is all you need. NeurIPS.
10. Wang, Y., et al. (2021). CodeT5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation. EMNLP.
11. Brown, T., et al. (2020). Language models are few-shot learners. NeurIPS.
12. Ray, B., et al. (2022). On the security of modern AI-based code generators. ACM CCS.
13. Austin, J., et al. (2021). Program synthesis with large language models. ICLR Workshops.
14. Gupta, R., et al. (2017). Deep learning approaches for code modeling. ICSE.
15. Svyatkovskiy, A., et al. (2020). IntelliCode: AI-assisted development. ICSE.
16. Alon, U., et al. (2019). Code2vec: Learning distributed representations of code. POPL.
17. Rabinovich, M., et al. (2017). Abstract syntax networks for code generation. ACL.
18. Nguyen, A. T., et al. (2013). Statistical learning approach for code suggestion. FSE