International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 12 Issue: 06 | Jun 2025 www.irjet.net p-ISSN: 2395-0072

ARTIFICIAL INTELLIGENCE IN PROGRAMMING: TRANSFORMING LANGUAGE STRUCTURES AND CAPABILITIES

Kamil Ansari[1], Dr. Peeyush Kumar Pathak[2]

[1]*Master of Technology, Computer Science and Engineering, Goel Institute of Technology And Management, Lucknow, India*
[2]*Assistant Professor, Department of Computer Science and Engineering, Goel Institute of Technology And Management, Lucknow, India*

Abstract - *Artificial Intelligence (AI) has grown so fast and its impact has had a great role in shaping and use of programming languages that have not only affected the structure of the programming languages but also of the developers. The given paper dwells upon the notion that AI reshapes classical codes paradigm by bringing automation, natural language processing, intelligent code generation, and adaptive environments of developing. The study started with a historical perspective of tracing the path of symbolic reasoning to the formation of the current AI-assisted tools including GitHub Copilot and the OpenAI Codex. In comparative analysis, we look at the performance and efficiency of the AI generated code as compared to traditional human coded solutions through the realms of web development, mobile applications and game design as some of the major domains in which we analyze such initiatives. Empirical findings suggest that AI helps to reduce the amount of time it takes to develop, decrease the occurrence of syntactic mistakes, yet cannot comprehend contexts and excel in the process of problem-solving most of the times. According to the finding, the future of programming is worked in the area of collaborative intelligence, whereby developers are helped by AI and not being replaced. This work helps to comprehend better the transformative role of AI in the sphere of programming and determine the possible perspectives of the rational and responsible application of the idea.*

Key Words: Artificial Intelligence, Programming Languages, Code Automation, Natural Language Processing, Software Development, AI-assisted Coding, GitHub Copilot, Code Efficiency.

1. INTRODUCTION

1.1 Background

History of programming languages can only be characterized by the constant evolution in the programming languages based on the necessity of a better abstraction, efficiency, and communication with humans. Whether it was primitive machine languages that were run with the binary commands to basic high-level languages such as FORTRAN, LISP and C, programming languages have been evolving over the years to fit the requirements of new technology. The invention of structural, object-oriented and functional paradigms further increased the space and flexibility of the expressions in the programming practices. At the same time, Artificial Intelligence (AI) which started as an area of study with the languages such as LISP and PROLOG has already become another dramatic change in software development [1].

Since the last few years, AI developed past being the topic of programming, and has become an accomplice in programming itself. Machine learning, the use of natural language processing and large language models (LLMs) like GPT-3 and Codex have allowed AI to contribute to writing, debugging and optimizing code. GitHub Copilot, DeepCode, and TabNine are tools that depict such a crossroad, as AI does not only assist software development but contributes to it directly. Since AI technologies are becoming more ingrained into the development context, they start affecting not only the design of the code but also the design of programming language to support and promote the capabilities offered by AI [2].

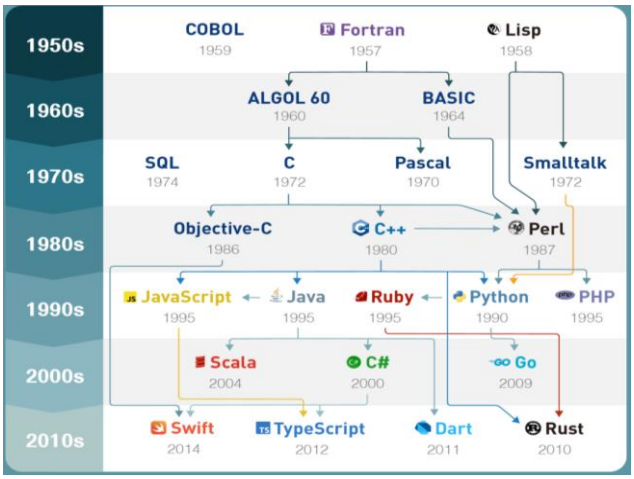

Figure-1: Evolution of Coding Languages.

1.2 Problem Statement

Even though the application of AI-based tools into the process of software development is becoming common, few empirical studies have been carried out to assess their

actual effectiveness in the configuration of programming language, productivity of developers and coding habits. Conventional methods of doing work basing on manual programming, debugging and refining are being confronted by the AI-aided systems that hold to be more fast-track and efficient in developing the work. Nevertheless, a loophole exists in the comprehension of the impact the tools have on the creativity, maintainability, and moral responsibilities of the developers. Questions have been raised on reliability, adaptability, and future impact of AI systems upon software engineering as a field as the systems start forming language syntax rules, produce large chunks of code, and start shaping design paradigms.

1.3 Objectives

In this research, the author is going to analyze the evolution of programming languages and software development processes that can emanate along with the incorporation of AI technologies. It pays attention to the determination of the way in which AI modifies structural paradigms of programming languages and how such changes affect cognitive and creative development of developers. The most important goal is to contrast a machine-generated code with one that people developed in terms of effectiveness, human-readability, scalability, and ethics. In so doing, the research will evaluate the benefits and the shortcomings of the usage of AI to aid the development process during the real-life example of coding such as of web, mobile, and game applications.

Also, the study aims to put the role of AI in context not necessarily in a capacity of the tool used, but as the partner in the development. This entails the examination of AI systems on learning abilities and the behavior of the system and its effect on human judgment during the codes. Finally, this study is going to merge into a complex framework to assess the changing role of AI in the programming context and present the recommendations of its successful and artless integration.

2. LITERATURE REVIEW

2.1 AI and Programming Languages

These two are the merge of artificial intelligence and programming language that over the years has evolved. However in the initial phases AI was much as something that can be programmed but not as something that takes part in the system of programming. Some languages were even designed to be particularly geared toward symbolic reasoning and logical inference like the LISP and PROLOG languages [20-21]. In 1958 LISP was introduced which was known specifically to be able to manipulate a symbolic computation and also recursion as well as list processing, and thus was popular with AI research in the subsequent decades. In the same way, PROLOG using the formal logic, provided declarative style of programming in which relationships and facts were important rather than sequential instructions. These languages embodied the intellectual structure of the early AI: rule-based, deterministic and firmly rooted in the mathematical logic [3] .

Due to the development of machine learning and deep learning, in particular, the point of view planned by AI went in the direction of data-driven models. The innovation of the current AI especially the large language models such as GPT-3 and Codex created by OpenAI shifted the paradigm in which AI communicated in programming languages [22]. Now, trained with huge code and natural-language corpora, such models have been able to learn to produce syntactically valid and functionally meaningful tuples of code in situ [4]. The opportunities of AI to assist in writing are already seen in such tools as GitHub Copilot, which is powered by Codex and has already proven itself in its growing ability to process not only meaning but also hint at the use of refactoring and codebase improvements [18-19]. Such transition will entail a change in how AI is used so that it is no longer the tool that is programmed, but a helpful and intelligent partner in programming with the ability to exercise a measure of control over the structure of language, design and even flow of work in software development [5-6].

2.2 Key Theories and Models

Behind the development of AI in programming are rival theoretical models that describe the strengths as well as the weaknesses of the technology. Among the most important dichotomies there is a symbolic versus connectionist approaches dichotomy. Symbolic Artificial intelligence, which was most prominent during the mid 20 th and late 20 th century is based on deterministic representations of knowledge and logic-based reasoning. Such languages as LISP and PROLOG were developed based on this paradigm where the main focus was made on structured knowledge, rules, and deduction [7]. On the contrary, the present-day AI breakthroughs rely on connectionist models, including artificial neural networks [12]. They are flexible to process the language, vision, and other complex inputs without having established rules to learn, because in them the patterns are taught by means of data [8].

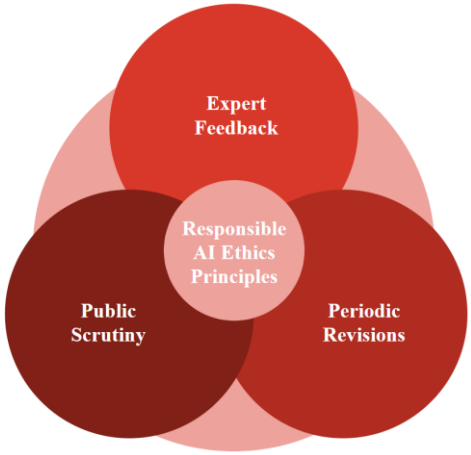

Figure-2: Principle of AI

Founded on linguistics and cognitive science, the so-called sound symbolism and language evolution theories have been introduced also as the explanations of the way AI models learn to use and acquire the language-like structures [11]. Research indicates that artificial intelligence (as is the case of a human learner) might learn phonetic patterns and symbolic associations without any actual rule. More so, the concept of convergence of language, or that AI starts to resemble the communication of a human being, has finally picked up with the emergence of NLP-driven models [9]. Such models can now respond in a humanlike way, can adjust the intonation and syntax, can even engage in a context-specific conversation. When translated to the realm of code, such capabilities would turn into AI capabilities that would not just rewrite the syntax, but start to understand semantics and design intent as well, and would be closer to how human developers think of the problem [10].

2.3 Gaps Identified

Though the theoretical and technical progress is quite significant, some gaps in research could be observed in the sphere of AI and programming. A gap in studying empirically the direct comparisons between the traditional human-centric coding and AI-assisted or AI-generated programming in real-world applications (in the real environment and not on simulated environments) might be one of the most urgent ones. Though there are some anecdotal reports and case studies showing the efficiency increase of using such tools as GitHub Copilot, there have been no rigorous quantitative studies of the quality, reliability, and maintainability of the code generated by AI tools in general or in specific application domains (such as web development, mobile platforms, and embedded systems). Also, the impact of AI solutions on the cognitive load of programmers, learning curves of the newbies, and long-range consequences of such patterns on the education and best programming practices remain poorly analyzed [13-14].

In addition, most of the current researches focus on the potential strength of AI without mentioning the disadvantages and ways in which it can be abused. Algorithms as well as AI generated codes, have ethical considerations including algorithmic discrimination, explainability as well as accountability that are normally given in theory but are hardly ever put to test in practice. Lack of critical and comparative analysis hampers us in learning how AI is moving programming in radically new directions than those provided by the surface level automation capabilities of AI. To do so, the gaps in the literature should be addressed and form the framework of a more nuanced and evidence-based approach suitable to direct the future research, tool development, and responsible use of both research software engineering and AI in software engineering.

3. METHODOLOGY

3.1 Research Design

The design of the research is comparative and of a mixed-method type as it would combine both a qualitative and a quantitative method to determine the consequences of the introduction of artificial intelligence into the contemporary programming process. The reasoning behind such a design is that besides measuring the performance variables like speed and accuracy, there is also the need to make the subjective observations that pertain to the elements of usability, adaptability as well as the experience of the developers [15]. The qualitative part is used in order to learn the contextual and experiential aspects of the AI-assisted programming, whereas the quantitative part is utilized in order to measure the empirical results such as the frequency of errors, their time effectiveness, and codes robustness. With this bilateral framework, there is a change of perspective that balances the analysis of AI and the conventional programming approaches to give a better insight into their positive and negative attributes.

3.2 Tools and Platforms

The experimentation, which is the basis of research, is conducted in real life either through AI-based or conventional programming environment. In the case of the AI-assisted programming, the tools like GitHub Copilot, ChatGPT, and DeepCode were used. GitHub Copilot works with OpenAI codex and supports IDE code suggestion in real-time. ChatGPT was applied when creating code snippets and natural language processing and DeepCode provides intelligent static code analysis and on-the-fly suggestions on how to improve the quality of software [16]. On the old school, work is done on React Native as a mobile app, HTML/CSS/PHP as a web development platform and Python integrated development environments like the PyCharm to test and debug code by hand. The tools are widely-used platforms in AI-enhanced

and traditional program coding space, which makes it fair to compare it with various projects types.

3.3 Evaluation Metrics

To quantify efficiency and performance of both the AI-assisted and conventional techniques of coding, a number of criteria of evaluating were introduced. Those are code efficiency (measured as compactness and runtime of the code), error rates (measured as number of syntax and logic errors identified during the process of execution or review), development time (measured as timestamps between starting the process and a successful deployment), and adaptability (how easy it is to purport the code to other tasks or platforms). The criteria provide a multidimensional concept of the software quality and an individual programmer efficiency.

Table-1: A comparative summary of metrics.

Metric	AI-Assisted Coding	Traditional Coding
Code Efficiency	High for basic tasks, moderate in logic-heavy functions	High in complex structures, lower in rapid prototyping
Error Rate	Low in syntax, higher in logic flaws	Moderate, mitigated by experience and debugging
Development Time	Significantly lower	Higher, especially in large projects
Adaptability	Medium, often requires developer intervention	High, especially for experienced programmers

3.4 Case Scenarios

In order to test the above metrics in a real-life scenarios, the following are the three different situations that were chosen in which software development is involved; web development, mobile application, and game development. The web development example relates to the development of a "Contact Us" page in HTML, CSS, and PHP, where the content was created with the help of AI, in the form of ChatGPT, and layout propose suggestions were made. The developed mobile app consisted of a form that was simple to create on the basis of React Native and required a user to enter data into it and submit it, with AI tools giving both content and a code snippet at the component level. The game development case concentrated on a traditional Snake game made using the React Native. Both methods and AI were used in both instances to generate something that worked just the same, which is why the performance across methods could be compared head to head.

The selected case scenarios are likely to be encountered during the practice being of practical value and having a variety in terms of coding complexity, models of interaction and platform-specific requirements. They allowed an all round comparison of the usefulness of AI in various situations of programming, ranging as far as the basic UI generation or the engaging logic and interactivity with the user.

3.5 Data Collection and Validation

A set of observation logs, performance Benchmarks and human expert reviews had been used in the collection of the data. In every coding session, we kept logs in order to record the duration of development, alterations of codes and rectification of errors. There were also benchmarking tools in order to determine run times and responsiveness of codes especially in the mobile and game development activities. Further, experienced programmers were able to revise each codebase blindly (without knowing it was generated by either AI or a person) and leave qualitative feedback on the aspect of maintainability, readability, and logic behind the design.

The validation was provided by performing several runs per task on various environments and with the provision of similar baseline objectives. Triangulation of feedback using the logs, benchmarks, and reviewers was done so as to make valid conclusion. The way of implementation is transparent, can be reproduced, and has practical importance, which places the study as valid technically and based on software development challenges in the real world.

4. RESULTS

4.1 AI-Generated vs. Traditional Code

As evidenced by the conducted comparative study of AI-generated and traditional human-written programs, significant distinctions could be observed along the main parameters, such as the speed, the number of errors, and the complexity of codes. According to the rapidity of development, AI-based devices like GitHub Copilot and ChatGPT saved much time when working on monotone and boiler-plate editing. It is possible that developers who code with the help of these tools can perform functions of code segments more rapidly than developers who code manually mainly because the AI allows auto-complete of structures, library suggestions, and work on syntactic formatting with a limited contribution of human beings.

But as regards the error rates, there were differences in results based on the complexity of tasks. Although AI-assisted code had fewer syntactic bugs because of the manner in which it was modeled using large sets of correct code, it sometimes injected logical errors, particularly where it performed tasks that demanded contextual intelligence or non-sequential programming. Conversely, the traditionally coded programs were slower to code with, but displayed greater logical correctness and lesser

runtime faults, mostly because of the human instinct and debugging know-how.

When it came to code complexity, it was also observed that AI solutions were dense or over-generalized in nature which although did work, were not very clear and modular in structure as they would be had they been written by a human. This made the script a little less maintainable or readable, especially in more advanced logic based applications like games. On the whole, AI-assisted coding can perform much better than conventional coding when it comes to designing prototypes and syntax control, but when it comes to edge cases, subtle logic, traditional coding appears to be more frugal.

4.2 Task-Specific Findings

The paper had implemented three case scenarios- development of web, mobile application and implementation of game logic to analyze AI-based and regular programming methods within the specified conditions.

Both AI and conventional computing methods performed similar tasks in the challenge of developing a web. The completed code with the help of AI was about 40 percent quicker and was semantically in its words and HTML and CSS structures. The created content and layout were not only beautiful and artistically pleasing but also professionally accurate, which means that AI tools would be perfect to make an ordinary web application based on UI. The traditional coding was also slower but with customization and provided better control on form validation and server side scripting with the use of PHP.

In the case of mobile app development, AI proved to be a lot faster, having done the task with almost 50 percent speed of a manual coder. Nonetheless, the use of AI to compile the React Native code produced problems when edge-case scenarios were added, including output validation without being completed and incorrect use of APIs along with its failing. There was need of human intervention where developers had to manually ensure that such cases were dealt properly which means that the anticipating powers of the AI was not so competent. Conversely, the human-coded one proved to be more trustworthy in a larger scope of the test cases, however, time- and effort-consuming.

In this case the white world scenario was comprised of a game development scenario, where we were expected to use a popular Snake game, that is why the classic game of Snake was developed with the help of React Native, and the traditional code was more successful than AI in the aspect of logical consistency and error control. Although AI might have created simple game loops and user interface, it was not able to produce coherent rules to manage collisions, scoring and to change game state. The logic that human-coded was more specific, modular and scalable and it stresses the necessity of experience related to the problem-solving skills in the case of working with logic-intensive tasks.

4.3 Quantitative Comparison

To support the qualitative findings, the following table summarizes key metrics measured across each task for both AI-assisted and traditional approaches:

Tanle-2: Quantitative Comparison.

Task	Time Saved (%)	Error Rate (AI)	Error Rate (Human)	Code Readability
Web Page	40%	Low	Medium	High (AI)
Mobile App	50%	Medium	Low	Medium (Human)
Game Logic	30%	High	Low	High (Human)

It is evident in the table that as compared to other projects that have the same requirement, AI can be applied to speed up the development process, in particular, in instances involving repetitive or UI-intensive activities. Nevertheless, the complexity of the topical tasks raises fewer doubts with regard to the lack of AI capabilities: logic, sensitivity, and vulnerability. In problem areas of the need of human perspective or profound problem solving logic, human programmers still offer the better error managing, flexibility and treasured structure design.

5. DISCUSSION

5.1 Interpretation of Results

The results of the research indicate the subtle perception of an ever-shifting position of artificial intelligence in the programming sphere. Among the most obvious results, it is worth highlighting the ability of AI to significantly increase developer productivity, mainly in the areas where any repetitive structure, user interface, and the generation of boilerplate code can be involved. With the help of GitHub Copilot and ChatGPT, developers can code at 50 percent faster in some domain areas of application including web and mobile development. These tools shorten the development life-cycle through the production of immediate recommendations, and minimized syntactic problems, and mechanical overhead build-up.

Nevertheless, this increase in productivity has a price to be paid since most of the high-end problem solving and issues that involve a high level of understanding over application principles or out of the box software

architecture are involved. In game development, as an example, the logic generated by AI often crashed when put in situations requiring multi-state logic or decision management situations or abstract cross-algorithms. AI can easily get the wrong context or produce a generic pattern that may bring logical inconsistency or poor design. These disadvantages indicate the existing weakness of the ability of AI to reason beyond its training set of data and underline the necessity in the software engineering with high level of complexity of the human intuition, knowledge, and creativity.

5.2 Ethical and Practical Considerations

Although efficiency is introduced with the help of AI in programming, there are also some important ethical and practical questions which should be discussed further. Among the list of the greatest concerns is the possibility of bias in AI-born code. The other issue is that models such as Codex and GPT-3 are trained on public code repositories, which can unwittingly transmit an existing bias or other undesirable trends or practices or out-of-date coding conventions to the models. Such biases might be revealed in algorithmic decisions that are unfair or that may be applied in such sensitive areas as healthcare or finances.

Furthermore, the hallucinated code is a more recent issue of concern, a process during which the AI produces the code that is grammatically correct but either semantically incorrect or non-functional altogether. Such code may cause undesired behaviours (or security flaws) in situations when the output is not directly verified. The developers can trust the outputs of the AI excessively, at least in the situations when the code seems to be properly structured and technically sound.

Accountability is another burning problem. In the classical software development the source of a piece of code and its code logic can be attributed to particular developers or groups, so it can be clearly attributed. Code produced by AI devices tends to break the barrier between the work of a human and the work of a computer since you cannot say with certainty that some code was written by the one and not by the other. This makes it harder to attribute legal and professional accountability especially in a situation where the code is faulty and the result is data mining or failure within the system. With a growing involvement of AI in the code generation process, there is a critical need to develop well-defined ethical guidelines and norms that would provide some levels of transparency, validation, and supervision of the human responsibility to the overall development flow.

5.3 Human-AI Collaboration

This is because of all the advantages and the shortcomings of AI, the human-AI cooperation seems to be the most progressive one in the field of programming. Instead of defining an AI as something that could replace developers, it needs to be explained as an auxiliary tool that can help them be more productive, ease them the burden of mechanical work, and have more time to concentrate on innovation and architecture-level work. This symbiosis model envisages the work process in which AI will provide a code suggestion, complete routine logic, or assist in documenting the work, whereas human developers will review, optimize, and approve the outcomes.

The developers in this model have creative as well as ethical control of code creation making it careful, purposeful, and contextually aware on designer decisions and that, its decision in coding is centered on the greater system objectives. Human performance in terms of abstracting, innovating and contextualizing is beyond match. Simultaneously, AI also promotes the development process as it can act as a daily helper, learning specific coding patterns, proposing some improvements, and indicating the possible risks to a user.

Inclusion and openness in the process of coding development might also be provided through integration of AI into development and making it accessible and available to both beginners and non-programmers. The idea of developing software with the aid of natural language interfaces makes it possible nowadays to convert your idea into an actually working application even without having considerable knowledge in coding, expanding the demographic range in software development.

Table-3: complementary roles of AI and human developer.

Aspect	AI Role	Human Role
Speed & Efficiency	Suggests code, auto-completes routines	Reviews suggestions, integrates logic
Logical Depth	Limited contextual understanding	Designs core algorithms and workflows
Error Management	Detects syntax and common bugs	Resolves complex errors and logic gaps
Creativity & Innovation	Lacks originality	Invents new patterns and solutions
Accountability	No ethical responsibility	Holds legal and moral responsibility

6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

The paper has addressed the revolutionary nature of artificial intelligence on the various programming languages and developmental processes. The conclusions put emphasis on the fact that AI is not only speeding up the process of programming but also changing the face of writing, interpreting and using programming languages. The role of AI is changing due to such tools as GitHub Copilot and ChatGPT that help developers write code, optimize it, and debug applications in real-time. This development marks the beginning of a new stage in software engineering where AI can no longer be described as a passive tool in the back end but rather be seen as pro-active co- authored software creation. Notably, the paper also indicates that although AI has a significant number of strong points in connection to speed and automation, it is incapable of generating contextual intelligence and the creativity of logic design and decision-making.

The most hopeful one is the appearance of cross-breed intelligence, where AI does not cut off humanity, but complements its abilities. In such a paradigm, AI would be used to process repetitive and predictable work and human developers would be put to regulating architecture, innovation, ethical decision making and so on. This is because such a collaborative type guarantees effectiveness and efficiency, in addition to quality, innovativeness and responsibility in software systems. These two colluding intelligence can take the technological programming to greater heights of developing programs that are far much easier, productive, and with life in the software development process.

6.2 Limitations

Although the present research has provided great knowledge, it had some limitations in regard to the scope. First, the study mostly dealt with frontend work of web development, mobile apps, and somewhat game algorithms, which, at the same time, is sound and quite applicable, but it does not cover all possible fields of software engineering. It was only focused on front-end development, database integration, systems programming, and applications of enterprise scale; therefore it is one of the areas where performance of AI is not examined. Accordingly, the certain solutions made here will not be universally applicable to any form of development duties.

Second, the AI models applied in the study (such as GitHub Copilot and ChatGPT) are ones that get trained on publicly available code and are accurate, biased, and more or less diverse in domain to an extent. This only restricts the nature of generalisability of results particularly in highly specialized, or niche areas where these models might not be exposed. In addition, applications in the real world deployment scenarios of scalability, security, and maintainability were hardly touched upon since the major testing was performed in the controlled development environments. This has presented some limitations implying that there should be more empirical applications in production level settings.

6.3 Future Scope

With the current progress on the development of AI and its active involvement in the development of programming, a number of possible perspectives of the further studying and implementation become apparent. The representation of one of the directions is the use of AI in the design of compilers. The traditional compilers concentrate on syntax and rules of optimization, however, the compilers driven by AI may learn how to optimize adaptively, find inefficiencies, and even suggest alternative syntactic forms to achieve performance improvements. This type of compilers would in effect fill the gap that currently exists between high-level specifications and implementation.

The other exciting direction is the ability to create individual programming AI assistants, which learn about the particular developer, their style, preferences, and coding history. These assistants would not be the generic AI tools, but could provide specific suggestions, use consistent design patterns and even suggest future code requirements, based on the user habits. This would greatly increase productivity especially in an area of a team effort where continuity and stylistic consistency is a factor.

REFERENCES

1. Ahmad, Aqueel et al. “Unified Pre-training for Program Understanding and Generation.” Proc. NAACL-HLT ’21, 2021, pp. 2655–2668. (dl.acm.org)

2. Alon, Uri; Sadaka, Roy; Levy, Omer; Yahav, Eran “Structural Language Models of Code.” ArXiv preprint, Sept. 2019. doi: 10.48550/ARXIV.1910.00577. (arxiv.org)

3. Chen, Mark et al. “Evaluating Large Language Models Trained on Code.” ArXiv preprint, July 2021. doi: 10.48550/ARXIV.2107.03374. (ouci.dntb.gov.ua)

4. Cunningham, Cody et al. “GPT-3 Few-Shot Learning for Code Explanations.” SIGCSE ’22, 2022, pp. 551–557. doi: 10.1145/3478431.3499370. (dl.acm.org)

5. Feng, Zhangyin et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” ArXiv preprint, Feb. 2020. doi: 10.48550/ARXIV.2002.08155. (ouci.dntb.gov.ua)

6. Ferrario, Paolo; Winter, Eugen “Applying Human Values Theory to Software Engineering

Practice." IEEE Trans. Software Eng., vol. 49, no. 3, pp. 973–986, Mar. 2023. doi: 10.1109/TSE.2022.3170087. (arxiv.org)

7. Finnie-Ansley, James et al. "The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming." ACE '22, 2022, pp. 10–19. doi: 10.1145/3511861.3511863. (dl.acm.org)

8. Floridi, Luciano; Chiriatti, Massimo "GPT-3: Its Nature, Scope, Limits, and Consequences." Minds and Machines, vol. 30, no. 4, pp. 681–694, Oct. 2020. (dl.acm.org)

9. Guo, Ensheng et al. "Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study." ArXiv preprint, Sept. 2023. arXiv:2309.08221. (arxiv.org)

10. Guo, Philip "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education." SIGCSE '13, 2013, pp. 579–584. (dl.acm.org)

11. Hellendoorn, Vincent et al. "PolyCoder: Large Open-Source Model for C Code Generation." Proc. ACM SIGPLAN Mach. Program. '22, 2022. (dl.acm.org)

12. Helmuth, Matthew; Kelly, Peter "PSB2: The Second Program Synthesis Benchmark Suite." Proc. GECCO '21, 2021. doi: 10.1145/3449639.3459285. (dl.acm.org)

13. Idrisov, Bulat; Schlippe, Tim "Program Code Generation with Generative AIs." Algorithms, vol. 17, no. 2, p. 62, Feb. 2024. doi: 10.3390/a17020062. (mdpi.com)

14. Imai, Saki "Is GitHub Copilot a Substitute for Human Pair-Programming? An Empirical Study." Proc. ICSE '22 Companion, 2022, pp. 319–321. doi: 10.1145/3510454.3522684. (dl.acm.org)

15. Kanade, Aditya et al. "Learning and Evaluating Contextual Embedding of Source Code." ICML '20, 2020, pp. 5110–5121. (pmc.ncbi.nlm.nih.gov)

16. Khan, Junaid Younas; Khan, Ghulam Ullah "Automatic Code Documentation Generation Using GPT-3." ArXiv preprint, Sept. 2022. doi: 10.48550/ARXIV.2209.02235. (arxiv.org)

17. Liu, Deyu et al. "Multi-task Learning Based Pre-Trained Language Model for Code Completion." ASE '20, 2020, pp. 473–485. (pmc.ncbi.nlm.nih.gov)

18. Mohammadi, Behrad "Pel, A Programming Language for Orchestrating AI Agents." ArXiv preprint, May 2025. arXiv:2505.13453. (arxiv.org)

19. Niu, Rui et al. "CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot." USENIX Security '23, 2023, pp. 2133–2150. (arxiv.org)

20. Ochs, Moritz; Narasimhan, Karthik; Mezini, Mira "Evaluating and Improving Transformers Pre-Trained on ASTs for Code Completion." IEEE SANER '23, 2023, pp. 834–844. (arxiv.org)

21. Olausson, Leo et al. "Demystifying GPT Self-Repair for Code Generation." ArXiv preprint, June 2023. arXiv:2306.09886. (arxiv.org)

22. Ombredanne, Philippe "Free and Open Source Software License Compliance: Tools for Software Composition Analysis." Computer, vol. 53, no. 10, pp. 105–109, Oct. 2020. (arxiv.org)

23. OpenAI "GPT-4 Technical Report." ArXiv preprint, Mar. 2023. arXiv:2303.08774. (arxiv.org)

24. Parvez, Nabil et al. "Retrieval Augmented Code Generation and Summarization." Findings of ACL EMNLP '21, 2021, pp. 2719–2734. (arxiv.org)

25. Pearce, Hammond et al. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions." IEEE S&P '22, 2022, pp. 754–768. (arxiv.org)

26. de la Torre, Javier "From Tool Calling to Symbolic Thinking: LLMs in a Persistent Lisp Metaprogramming Loop." ArXiv preprint, June 2025. arXiv:2506.10021. (arxiv.org)