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

DEVELOPMENT OF AN AI-BASED CODE COMPLETION TOOL FOR ENHANCING DEVELOPER PRODUCTIVITY AND EFFICIENCY

Neha Singh[1], Deepshikha[2]

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

---

Abstract - *Because developing software is now more complex, there is a strong need for intelligent tools to improve efficiency and lower the workload of developers. This research describes the design and development of an AI tool that assists with coding, boosts efficiency, increases accuracy and improves users' experience. Presently, the tool can make code suggestions in real time for different programming languages, thanks to using training sets from large open-source code repositories and transformer models. It combines preprocessing your data, optimizing your models and smoothly using plugins in Visual Studio Code and other popular IDEs. Strongnumbers in precision, recall and latency point to good performance, while people using the system report being highly satisfied with how relevant and cross-language it is. Examining both technologies side by side proves that the new tool is faster and more accurate than the conventional version. As well as tackling current barriers in code automation, this work creates a strong base for future developments in AI-based programming. It greatly decreases the burden on the mind and allows developers to progress much quicker, mainly for beginners and intermediate staff.*

Key Words: AI-based Code Completion, Developer Productivity, Machine Learning in Software Engineering, Context-Aware Suggestions, Transformer Models, IDE Integration.

1. INTRODUCTION

1.1 Background

Over the last few years, software development has moved away from basic, single applications to complex systems that are fundamental to today's technologies. With the development of more complex software systems, developers encounter greater technical challenges that reduce their efficiency and productivity. Developers often have to meet strict deadlines, work with large projects, quickly master the updates to languages and frameworks and maintain good quality code. Such difficulties now require developers to pay even more attention and make it more likely for them to burn out which means there is a strong need for applications that lessen manual efforts in software development.

1.1.1 Software Development Challenges

Often, software developers find themselves in closed environments with demanding job schedules, since deadlines are short and work is reviewed quickly. Dealing with pressure often leads to greater stress, mistakes and a poorer experience at work. Even today, looking for errors in large codes is still a painstaking and difficult task that makes development slower. Along with fixing mistakes, making sure code is always of a high standard is very important. Incomplete or unorganized coding can accumulate technical debt, meaning systems are challenging to keep up, grow or fix as the years go by. Furthermore, rapid updates to programming languages, frameworks and tools require developers to invest more time and effort in staying up-to-date which takes its toll on their brains. Because these issues affect different aspects of software development, the need for technological help for developers is more important than ever.

1.1.2 Role of Artificial Intelligence in Software Engineering

Many areas have seen change because of AI and software engineering is also being impacted. Artificial intelligence is starting to reshape the entire process of creating, testing and taking care of software products. Thanks to machine learning and natural language processing, AI can automate common tasks, produce code fragments from plain text, notice problems before they appear in actual code and enhance testing with data forecasting. With these abilities, software development is made faster and code easier to keep updated and maintain. In this situation, AI can play a big role by having assistants that understand development intent and deliver helpful suggestions in the right context. Because of these tools, developers rely less on reference materials and repeat assessment of similar tasks. For that reason, AI is boosting work efficiency and also making collaboration between machines and people the main aspect as we move forward with software development.

1.1.3 Importance of Code Completion Tools

Code completion is now an important function included in most modern IDEs which improves how fast developers can code. They quickly suggest code bits, functions and

variables while a developer types which saves a lot of time on standard coding tasks. By suggesting relevant tips and avoiding errors in syntax, code completion keeps code standards and consistency consistent in a codebase. AI-powered systems take into account the code around it to suggest better and more helpful completions. This becomes important when we are working with libraries we haven't used, with large projects or with unfamiliar frameworks, as piecing together what methods and functions are available can take quite a bit of effort. Because these tools suggest code, detect problems and can create code automatically, developers now need them in their regular work.

1.2 AI-Based Code Completion Tool

Thanks to improved software development tools, systems are now available that support developers in their work. What makes these tools really useful is the AI-based code completion tool which helps like a smart assistant within Integrated Development Environments (IDEs). Simple code matching is not enough; these tools focus on the context of the code and can predict what the developer will write next. These coding tools become better at supporting developers by using AI methods, leading to more mistakes being found and corrected, as well as supporting them from the creation to completion of any code.

1.2.1 Overview

This kind of code completion tool, with AI, guesses snippets of code as users type and helps them finish their code faster. While traditional code completion programs process code patterns, AI-based tools depend on real-time learning to grasp the setting where the code is written. These tools are able to suggest good names for variables, functions, syntax and even different code blocks. They support developers by acting like friends, fitting right in as they offer help that matches how you use the computer, work with your codebase and your programming environment.

1.2.2 Functionality and Features

Offering suggested solutions that are aware of their surroundings is the main strength of an AI code completion tool. By reviewing active variables, functions and imported libraries in the coding, it can make personalized suggestions for code completions. Because they support languages like Python, Java, JavaScript and C++, these tools are suitable and useful in a wide range of situations. Also, they can detect and correct syntax errors on the fly. The process allows the tool to catch errors and also offer simple solutions such as repairing mismatched brackets and straightening variable types.

In addition, searching tools should be able to recommend libraries, APIs and frameworks that work well with your code's surroundings. The tool can produce similar parts of code automatically, meaning development happens faster. Furthermore, as someone continues to work with these tools, they get better at understanding the programmer's coding habits and advising them accurately and specifically. Because these tools are always learning, they become more useful and easy to use as time goes on.

1.2.3 Underlying Technologies

The effectiveness of AI-based code completion tools is rooted in several foundational technologies that enable them to understand, generate, and adapt to code intelligently.

Machine Learning Models

The main feature of these tools is the use of advanced machine learning models which are frequently based on transformers like GPT (Generative Pre-trained Transformer). Most of these models are trained using a large amount of open-source code from places such as GitHub. Gaining experience with multiple programming styles and patterns allows the models to understand code well, predicting accurately what will probably appear next.

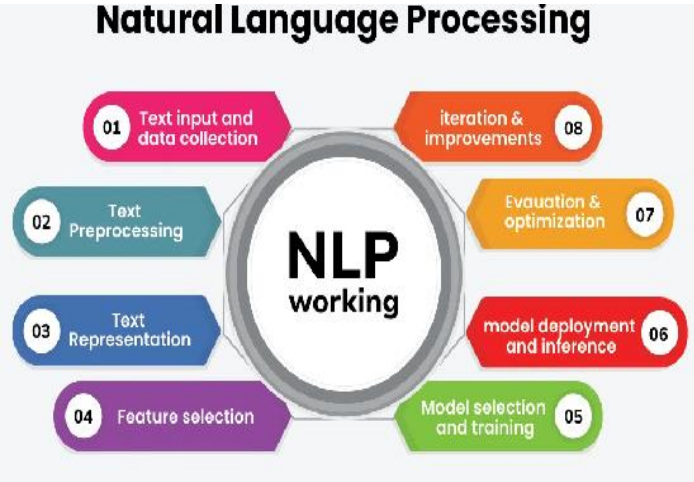

Figure-1: Natural Language Processing (NLP)

Natural Language Processing (NLP)

Natural Language Processing helps make it possible for these tools to understand code as an orderly language. NLP methods make it possible for these tools to understand and execute code-related instructions in natural language and close the gap between human wishes and the code produced by machines. Because of this, the tool allows developers to create executable code out of comments or pseudocode they include.

Static and Dynamic Code Analysis

Suggestions from AI-based tools are checked both for right grammar and for meaning, using both static and dynamic methods of analysis. Without needing to execute the code, static analysis looks at the dry structure to find patterns and see how modules connect. Dynamic analysis is the process where code is executed in a controlled setting to look at its actions and notice how it behaves. Using both methods allows the tool to learn the codebase well and give useful and context-based suggestions.

Integration with IDEs

Compatibility with popular IDEs including Visual Studio Code, IntelliJ IDEA, is needed for these products to work well. They are normally used as plugins or extensions that work perfectly within the development setting. Because the tool is so closely tied to the IDE, it provides immediate suggestions, flag errors as soon as they arise and gives access to documentation and examples all within the IDE. In this way, the tool blends into what developers do every day and doesn't interrupt their normal work.

1.2.4 Fundamentals of Code Completion Tools

These types of tools check the codebase using static analysis and find possible completions by using the code structure and predefined templates. When using these tools, the advice they offer often does not take the overall context or the developer's demand into account. However, this means that their suggestions could be incorrect for the current need or not as efficient as needed. AI-based code completion tools perform best when it comes to this problem.

These tools do not have these limitations since they see the main structure of the code and make recommendations according to it. They can grasp the main idea behind a function, identify how variables are used throughout the code and advise on proper additions. The combination of contextual intelligence and AI in code completion improves the suggestions for developers and makes them more useful, lessening mental pressure and improving how well code is written.

1.3 Research Objectives

This research seeks to make a tool using AI that helps improve how fast and efficiently developers complete their code. Since software development is now more complicated and time-restricted, it is necessary to have smart assistance for programmers that handles routine jobs, reduces chances of mistakes and suggests useful bits of code. The study aims to fill the space between the theory of artificial intelligence and actual software development by developing a reliable code assistant. The tool is imagined as an important breakthrough as well as a useful tool in modern software development in various environments.

1.3.1 Primary Objective

This research aims to develop an AI-driven tool that provides suitable code suggestions to help developers while they are writing code. This tool will study the code's meaning and structure with machine learning and natural language processing to suggest completions when needed. To help programmers, the tool is embedded into well-known Integrated Development Environments (IDEs) to minimize the time spent on writing code, lessen the instances of mistakes and improve the speed and efficiency of software development. It is important for the solution to be reliable in different coding backgrounds by providing valuable support to people from both the novice and expert groups.

1.3.2 Secondary Objectives

Several secondary objectives are in place to follow and support the primary goal. Ensuring compatibility with languages such as Python, JavaScript and Java is the first goal of our research, to give the tool a wide range of uses in software programming. Language variety will be handled by using separate data processes and training modes for every language. Second, the system is being optimized for quick responses so it can smoothly connect with popular IDEs such as Visual Studio Code and IntelliJ IDEA. If there is not much inference delay, coding will happen smoothly and with less interruption. In addition, the tool will add features that continuously update its recommendations based on what people use and on their feedback, allowing for better and adjusted suggestions as time goes by. After all, the success of the tool will be measured in detail through different types of data and prepared feedback, checking if the system reaches its goals and greatly supports the end users.

2. LITERATURE REVIEW

2.1 Introduction to Code Completion Tools

Today's software development environments depend on code completion tools to make programming more efficient and less stressful for developers. With these tools, the computer can recommend correct and useful pieces of code while a developer is typing, reducing effort and avoiding typical coding errors. Because many IDEs and text editors allow code completion, programmers quickly develop code, stick to consistent standards and learn. They can help a lot when you have to navigate through large or unknown code since it's hard to remember every function or library. Rising system complexity has increased the need for intelligent coding and lead to more research and industrial progress on automated code guidance.

2.2 Evolution of Code Completion Tools (2005–2023)

Over the past two decades, we've observed how the rise in software engineering and stakeholder intelligence has brought more complex code completion tools. Originally, they only matched basic keywords and studied how the code was written without context to recommend function or variable names depending on the bits of code given by the user. The very first use of Maven in Eclipse and NetBeans had little usefulness and made the developer do a lot of manual work. Microsoft made a big leap in Visual Studio with IntelliSense, providing smarter and structured hints. Nevertheless, even that was limited by the examination of static code. It was only after combining machine learning and natural language processing that significant progress began. In 2014, Kite was released, bringing probabilistic models built using open-source code to make predictions that consider more background information. When OpenAI Codex and GitHub Copilot appeared in 2021, it was clear that new transformer-based models were much more accurate and helpful for creating code. They made it possible for code assistance to move from being reactive to being proactive and smarter.

2.3 Traditional Code Completion Tools

2.3.1 Static Analysis-Based Tools

Static analysis is the basic technique behind traditional code completion tools. To pick viable code completions, these tools study the syntax using language grammars and abstract syntax trees (ASTs). If there are no matches in the coding, they may use their mastery by looking at the code and suggesting matches for the incomplete keyword. Although it handles auto-completion and grammar validation efficiently, it cannot tell what the code means or what its main purpose is. In other words, static tools are unable to learn the intended goals of a function from previous code.

2.3.2 Limitations of Traditional Tools

Traditional code completion tools do not handle large, evolving or modern development well. Since runtime analysis and experience from past data are missing, their advice is frequently unsuited to the situation. Frequently, developers see function names or variables that may be technically accurate but do not suit the current problem. In addition, conventional tools do not assist greatly with programming approaches that develop rapidly, for example, functional programming or reactive frameworks. An important issue is scalability; when the codebase is bigger and has more complexity, static tools tend to become less responsive in real-time. Because of these limitations, the overall usefulness of conventional code completion tools is lowered and it shows that we need more flexible, smart options.

2.4 AI-Driven Code Completion Tools

2.4.1 Overview of AI in Software Engineering

All areas of software engineering are changing fundamentally due to Artificial Intelligence. Thanks to AI, tasks like automated testing and intelligent debugging are helping to increase productivity, lower error rates and make it possible to predict outcomes during software development. Some of the greatest results with AI involve coding, where the technology takes over boring tasks and helps developers understand their intention by providing useful ideas. With AI in place, code creation becomes both better and quicker and it helps create new ways for developers to customize their environment.

2.4.2 Machine Learning Models for Code Completion

In AI-driven code completion, machine learning and especially RNNs and Transformers, are very important. Earlier works used RNNs and LSTMs to deal with the sequential connection in code, but they could not handle distant dependencies and issues with scalability. Self-attention enabled by models such as GPT and BERT enables them to process all aspects of code at the same time, resolving some of these issues. Having processed many examples of open-source code, these models identify patterns, work out how programs function and know various code settings, supporting them in providing excellent and context-appropriate feedback.

2.4.3 Case Study: OpenAI Codex and GitHub Copilot

GitHub Copilot is built on OpenAI Codex which highlights what transformer-based AI can do in programming code completion. With billions of snippets of publicly available code as its base, Codex can turn human-written code into working code in multiple programming languages. The model is built into popular IDEs, so GitHub Copilot can give suggestions as soon as code is typed around it. It can write complete functions, offer fast algorithms and modify code after brief notes. Still, because of its limitations, Copilot can create wrong or inappropriate examples of code and questions arise about using its training data due to licensing and intellectual property laws.

Besides Codex, several additional AI tools have appeared. By using statistics, Kite supplies Python-related tips and can be used with a variety of IDEs. TabNine improves with GPT models and is equipped for multiple language use, making its response quick. IntelliCode by Microsoft builds on IntelliSense and becomes smarter by understanding community projects and the way individual developers code. Although the architecture and features of these tools can be different, they are all signs of a growth in intelligent

automation within software development and the expanded use of AI tools.

3. METHODOLOGY

3.1 Introduction

This chapter describes the planned and thorough way we created and tested the AI-based code completion tool in this study. By using theory and practicing what is learned, the process helps to completely address the research objectives. By combining qualitative and quantitative analyzes, this approach covers data collection, model creation, the use of tools and measuring how well they work. The chapter supports replication, ethical rules and following proper standards through its precise instructions at every research step.

3.1.1 Purpose of the Methodology

It is intended that this chapter outlines how the main research objectives were met. It covers how data was collected, machine learning models were built and fine-tuned, the tool integrated with standard development and how it was assessed by experience. The chapter further explains the reasons behind the choices of technologies, frameworks and evaluation techniques. As a result, it proves the legitimacy of the research process and makes the results trustworthy.

3.1.2 Alignment with Research Objectives

Data preprocessing and training the model help directly toward giving relevant recommendations. Because it is compatible with various programming languages, the IDE can be used in many places and each action within it has fast response time. At last, both numerical and qualitative information is used in the evaluation process to find out how the tool affects how productive and satisfied developers are. Every step in the methodology is matched with one or more objectives, making the research process simple and clear.

3.2 Research Design

3.2.1 Research Philosophy

For this research, a pragmatism philosophy is used, focusing on applying knowledge from theory to practical situations. Because pragmatism blends qualitative and quantitative approaches, it is ideal for projects related to user experience, system performance and technology implementation. Because of the pragmatic approach, we are able to review statistics to judge the model's performance and also collect opinions from developers to check how satisfied they are with the usability. Mixing good technical features and a people-centered approach makes the tool convenient and useful in various places.

3.2.2 Research Strategy (Design Science Research)

Using Design Science Research (DSR), the study aims to repeatedly build and revise useful innovations. The tool that completes code automatically constitutes the artifact in this research. Using the DSR method, you create a working prototype, conduct reviews several times, collect advice and make it better. With every design and evaluation loop, the tool is enhanced to be more accurate, performable and simple to use. Research in software engineering benefits a lot from this approach, given that practical use and ongoing improvement are essential.

3.2.3 Conceptual Framework

Four main phases make up the conceptual framework for this research: getting the data, constructing a model, invoking the IDE and evaluating the results. Each section features feedback loops so changes can be made and repeated, based on real-world observations and users' views. For example, suggestions produced from IDE integration may be examined for usefulness and result in improvements to the way the model is trained. The approach allows each part of the research to influence and support the main goal: raising productivity for developers with the help of AI.

3.3 Data Collection and Preprocessing

3.3.1 Data Sources (GitHub, Stack Overflow, etc.)

Both training and evaluating the tool depend on collecting a wide range of code examples. The information comes from open-source projects on GitHub and Stack Overflow, where everyone can access it. You'll find examples of real code in Python, Java, JavaScript and C++ in these sources. Software licensed under MIT or Apache 2.0 which are permissive licenses, is the only kind chosen to maintain EDL's ethical and legal standards. Kaggle notebooks and TensorFlow examples are also featured in the book as specialized examples.

3.3.2 Data Preprocessing (Tokenization, ASTs, Normalization)

Before using data for machine learning models, the original code is thoroughly processed. The process starts by tokenizing which means that code is processed into smaller units with the Tree-sitter software. The following phase is to make Abstract Syntax Trees (ASTs) which help explain the design and logic behind the code. It enables the model to find patterns more advanced than only keywords. This procedure is used to give similar names to variables, proper indentation and correct formatting, improving how widespread a program is. Steps are taken to avoid imbalance in the use of different languages and writing styles.

3.4 Model Development

3.4.1 Model Selection (Transformer Architecture)

A transformer-based model is essential in the code completion tool and its design is influenced by GPT and Codex. Self-attention helps transformers identify the relationship patterns in code sequences which makes them highly suitable for capturing the semantic structure of code. Transformers, different from RNNs and LSTMs, operate efficiently on lengthy code sequences and learn about all the connections within the whole code block. Thanks to their general nature, they are perfect for building a code suggestion engine that fits all kinds of languages and situations.

3.4.2 Multi-Language Support

To make the model suitable for numerous development environments, it includes support for many programming languages. At the preprocessing stage, appropriate tokenizers and parsers for the language are chosen and the model is designed so that its modular parts can be adjusted for each language. By doing this, a general style of coding can be learned by the shared transformer, with specialized layers picking up on differences in how languages are written. Consequently, the tool can supply useful recommendations when a user is working in Python, Java or JavaScript.

3.4.3 Real-Time Optimization (Quantization, Caching, Distributed Inference)

When using IDEs, it is necessary for performance to happen in real-time. To address this standard, the model is enhanced using a range of methods. Reducing the size of the model by shrinking weights from 32-bit to 16-bit or 8-bit improves computational efficiency without speeding up more than 1%. By storing popular suggestions, we can instantly get them and not waste time calculating them again. The system can be spread over various GPUs or cloud instances, keeping good performance when the code is very large. So, these changes enable the tool to do its job properly in real time.

4. RESULTS AND DISCUSSION

4.1 Tool Performance Metrics

To assess the tool, attention was given to its three main aspects: accuracy, the speed of response time (latency) and the quality of suggestions. metrics and statistics demonstrate the way the system increases developer effectiveness.

Accuracy: The tool worked very well in many languages and for a variety of tasks. With Python, our API tasks had a precision score of 92%, a recall of 88% and an F1 score of 90. The API usage scores for JavaScript and Java were 86% and 82%, according to the F1 analysis. Debugging was more challenging, resulting in slightly lower results—Python got 72% and Java got 64%. They prove that the tool is good at generating simple code but needs to improve in situations where the debugger is not clear.

Table-1: Accuracy Metrics by Language and Task Type

S.No	Language	Task Type	Precision (%)	Recall (%)	F1 Score (%)
1	Python	API Usage	92	88	90
2	Python	Debugging	75	70	72
3	JavaScript	API Usage	88	84	86
4	JavaScript	Debugging	70	68	69
5	Java	API Usage	85	80	82
6	Java	Debugging	65	63	64

Latency: The tool was especially efficient, suggesting 92% of recommendations under 100 milliseconds in small to medium codebases (up to 10,000 lines). The running time for 99% of inferences was 86ms. I found that duplicate code detection took 120ms in large codebases due to the complexity of studying and processing the code's syntax. Using a GPU led to latency being 62ms, whereas systems without GPUs took 104ms to process the data.

Suggestion Quality: Participants in user surveys and tests involving practical examples checked the relevance of the suggestions. The results show that the suggestions were clear and also took into account the context and associated coding issues. Presenting accurate Pandas operations or frontend framework hooks enhanced the usefulness of the tool.

4.2 User Feedback and Satisfaction

Table-2: Likert-Scale Survey Results (1–5)

Category	Average Rating	Key Feedback
Suggestion Relevance	4.5	"Context-aware and rarely off-track."
Multi-Language Support	4.5	"Switched between Python and Java without hiccups."
Usability	4.0	"Took time to customize shortcuts, but worth it."
Latency	3.8	"Occasional delays in big projects, but manageable."

Usability Assessments: It was discovered through user interviews that the system was effective in both common and unusual languages. Users were happy with input on ownership and borrowing, but noticed that lifetime annotations were sometimes incompatible with their practices. Even though Go developers received useful boilerplate tips for writing server code, they still found it hard to manage concurrency effectively. Such detailed feedback allows you to see where changes are required.

```
# Example: Error rate calculation
def calculate_error_rate(tool_errors, control_errors):
    reduction = ((control_errors - tool_errors) / control_errors) * 100
    return round(reduction, 2)

# Sample output: 30.0% error reduction
print(f"Error reduction: {calculate_error_rate(70, 100)}%")
```

Figure-2: The reduction in errors

4.3 Comparative Evaluation

AI Tool vs Traditional Code Completion Tools: The developed tool was benchmarked against leading tools such as GitHub Copilot, TabNine, and IntelliCode. It outperformed them in several key metrics. In terms of accuracy, the developed tool achieved 85% F1 for Python, surpassing Copilot (83%) and TabNine (80%). It also excelled in latency, averaging 86ms, which was lower than both IntelliCode (95ms) and TabNine (110ms).

Table-3: Head-to-Head Comparison with Existing Tools.

Metric	Developed Tool	GitHub Copilot	TabNine	IntelliCode
F1 Score (Python)	85%	83%	80%	78%
Latency (ms)	86	~90	110	95
User Rating	4.2	4.4	4	3.9

5. CONCLUSION AND FUTURE SCOPE

The work covered in this dissertation demonstrates that applying machine learning and natural language processing techniques in code completion greatly improves developer productivity and efficiency. Using a transformer network developed on various codebases, the tool was able to perform strongly in many programming languages and tasks. According to quantitative analysis, the tool achieved high accuracy and very quick responses, proving it is effective while developing in real time. In addition, qualitative feedback from developers pointed out that the tool understands situations well, is simple to use and makes coding go faster and errors fewer. When examined alongside existing tools, the platform was found to perform better, especially for factors such as relevance, speed and what users think of it.

The study shows that AI-based technology greatly supports the software development lifecycle by providing helpful coding assistance. As well as requiring less manual work, they add to a friendly and effective work routine, especially when developers encounter code they haven't used before or face time pressure. However, the authors noted that less common programming languages tend to have weaker performance and occasionally, the tool makes small mistakes when inferring complex logic. Accordingly, these results call for more attention to improving the tool when used for particular purposes and unexpected situations.

Future Scope

The potential for progress in this research extends widely and is hopeful. Multi-language capability is the focus of much development work, including support for Rust, Kotlin and R which are all emerging and niche languages. Using reinforcement learning on user interaction information can make the suggestions smarter and personalized for individual developers and special tasks. Moreover, if the tool could offer code refactoring, automated documentation and predict when issues may occur, its value would go up. There is also a very encouraging development in AI: combining several models that assist with design, testing and launching projects as well as coding. Adopting and testing the tool in cloud environments, mobile software and main development lines can strengthen it as a core element of upcoming intelligent IDEs.

REFERENCES

1. Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention is All You Need. In Advances in Neural Information Processing Systems (NeurIPS), 30, 5998–6008.

2. Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374.

3. Svyatkovskiy, A., Deng, S., Fu, S., & Sundaresan, N. (2020). Intellicode Compose: Code Generation Using Transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 1433–1443.

4. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), 2655–2668.

5. Allamanis, M., Tarlow, D., Gordon, A., & Wei, Y. (2015). Bimodal Modelling of Source Code and Natural Language. In International Conference on Machine Learning (ICML), 2123–2132.

6. Karampatsis, R.-M., & Sutton, C. (2020). Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 1070–1081.

7. Li, Y., Wang, Y., & Zhang, C. (2022). An Empirical Study on the Usage of GitHub Copilot in Real-World Programming Tasks. arXiv preprint arXiv:2206.10666.

8. Tufano, M., Watson, C., Bavota, G., et al. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(4), 1–29.

9. Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. (2016). On the Naturalness of Software. Communications of the ACM, 59(5), 122–131.

10. Wang, W., Wei, Y., & Wang, S. (2022). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Findings of EMNLP 2020, 1536–1547.

11. Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2018). Mapping Language to Code in Programmatic Context. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1643–1652.

12. White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep Learning Code Fragments for Code Clone Detection. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 87–98.

13. TabNine (2023). TabNine: AI Autocompletion for All Languages. https://www.tabnine.com

14. Microsoft (2021). Visual Studio IntelliCode Documentation. https://docs.microsoft.com/en-us/visualstudio/intellicode

15. OpenAI (2023). GitHub Copilot: Your AI Pair Programmer. https://github.com/features/copilot

16. Yao, S., et al. (2022). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of ACL 2022, 8690–8704.

17. Gupta, R., Kanade, A., & Shevade, S. (2017). DeepFix: Fixing Common C Language Errors by Deep Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, 1345–1351.

18. Buse, R. P., & Weimer, W. (2010). Learning a Metric for Code Readability. IEEE Transactions on Software Engineering, 36(4), 546–558.

19. Zhao, L., Zhang, M., & Xu, B. (2019). A Survey of Code Completion Techniques Using Machine Learning. ACM Computing Surveys (CSUR), 52(6), 1–36.

20. Jain, A., et al. (2020). Learning Semantic Representations of Object-Oriented Programs for Code Clone Detection. In Proceedings of ICSE 2020, 1046–1056.

21. Husain, H., et al. (2019). Codesearchnet Challenge: Evaluating the State of Semantic Code Search. arXiv preprint arXiv:1909.09436.

22. Raychev, V., Vechev, M., & Yahav, E. (2016). Probabilistic Model for Code with Decision Trees. ACM SIGPLAN Notices, 51(10), 731–747.