A Comparative Analysis of AI-Assisted Development Tools in Software Development

Marija Miloshevska[1] and Marija Kalendar[2]
[1]*MIR! Software Solutions, 1000 Skopje, North Macedonia*
[2]*Department of Computer Technologies and Engineering, Faculty of Electrical Engineering and Information Technologies, SS. Cyril and Methodius University in Skopje, Rugjer Boshkovikj Str. 18, 1000 Skopje, North Macedonia*
milosevskamarija99@yahoo.com, marijaka@feit.ukim.edu.mk

Keywords: Artificial Intelligence (AI), AI-Assisted Programming, Software Development, Frontend Development, Code Generation, Software Quality, Developer Productivity, Tool Comparison.

Abstract: Artificial Intelligence (AI) tools are increasingly integrated into modern software development workflows, assisting developers in tasks such as code generation, testing, and debugging. This study presents a comparative evaluation of three AI-assisted development tools - Cursor, GitHub Copilot, and Gemini - with a focus on their effectiveness in frontend development. The comparison was conducted through practical experimentation within a small-scale Vue 3 project, where the tools were assessed across representative tasks including component implementation, utility function development, and test creation. The findings indicate that AI-based tools can significantly improve productivity. However, the generated outputs frequently require manual review, correction, and refinement to ensure correctness and alignment with project requirements. Each tool demonstrated distinct strengths, but no single tool consistently outperformed the others across all evaluated tasks. The results suggest that AI tools are most effective when used as supportive assistants rather than autonomous replacements for developers. Future research may extend this evaluation to larger-scale projects, different frameworks, and more complex development scenarios.

1 INTRODUCTION

Software development has evolved rapidly over the past few decades, largely because businesses and consumers expect more advanced, reliable, and feature-rich software. As projects become increasingly complex, developers often struggle with tight deadlines, limited resources, and the constant need to maintain high code quality. To keep up with these demands, the industry has started embracing new and innovative approaches, and Artificial Intelligence (AI) has become one of the most influential drivers of this change.

AI technologies such as machine learning (ML), natural language processing (NLP), are changing how developers work across different stages of the software development lifecycle. Many tasks that once required significant manual effort can now be supported or automated by AI: code generation, testing, debugging, and deployment. Tools like OpenAI Codex and GPT-4 can turn natural language instructions into working code, suggest improvements, and even spot potential bugs. This not only saves time but also helps reduce human error, leading to more stable and dependable software.

As AI becomes increasingly integrated into software development workflows, its role is expanding beyond simple productivity gains. It has the potential to reshape how efficiency, accuracy, and innovation are defined in software engineering. Rather than replacing developers, AI is increasingly becoming a powerful assistant that helps them focus on more creative and high-level problem solving.

In this context, the study presented in this paper examines the practical effectiveness of contemporary AI-assisted development tools in frontend software engineering. Specifically, it provides a comparative evaluation of Cursor, GitHub Copilot, and Gemini within a controlled Vue 3 project environment. The tools were assessed through real development tasks, including component creation, utility function implementation, and test generation, in order to analyze their impact on productivity, code quality, contextual awareness, and overall usability. By systematically comparing their strengths and limitations, this paper aims to contribute empirical

insights into the role of AI as a collaborative support mechanism in modern software development workflows.

2 RELATED WORK

Artificial intelligence is being increasingly used in software development recently. Consequently, the topic has attracted a lot of research in the last few years. In the early days, AI was mostly used for automation or data analysis, but more recently the industry, as well as the research community, have increasingly focused on integrating generative AI and coding assistants directly into development environments. As a result, researchers are working on understanding the performance of these tools in real-world environments, and what are their limitations.

There have already been a number of research studies comparing AI coding assistants and evaluating how they perform. Results usually show that these tools do not behave the same in every situation. Some work better for structured programming tasks, while others are more useful when the developer interacts with the tool step by step. Experiments with programming problems show that AI can often handle simple tasks, but performance drops when tasks become more complex or require deeper reasoning [1] - [3]. This suggests that AI is becoming more and more helpful, but not something developers can fully rely on yet.

There are also studies that research generative AI in software engineering in a more general way. Generative AI is being used for testing, bug prediction, and code suggestions. Reviews that summarize research from recent years show that development and testing are the areas where AI appears most often [4], [5]. These works usually mention time savings and reduced manual effort as benefits. At the same time, they also point out issues like unclear model decisions, data quality problems, and difficulty fitting AI tools into existing workflows [6] - [8]. So, while the potential is clear, there are still some open questions.

Another topic that appears in the research literature is how developers feel about using AI tools. The verdict is that “trust is not automatic”. Developers tend to trust suggestions more when they are accurate and match the task they are working on. If suggestions are wrong or strange, confidence drops quickly [9]. This indicates that adoption depends not only on performance but also on user experience.

Some research work also focuses on how these tools are built and used in practice. For example, local code completion systems show that AI support does not always need cloud services to be useful [10]. This can matter for privacy and response time. At the same time, AI-based software brings new challenges, since models depend on training data and may change over time, which is different from traditional software behavior [11].

Overall, existing research suggests that AI-assisted development tools are steadily advancing and demonstrate high potential to support developers. However, a substantial portion of prior studies relies on benchmark evaluations or controlled experimental settings, which may not fully capture the complexity and variability of real-world software development environments. In practice, development workflows often involve changing requirements, contextual dependencies, and imperfect conditions that are difficult to replicate in laboratory settings. Consequently, there remains a need for empirical investigations conducted within realistic development scenarios. This study builds on prior work by presenting a practical, task-oriented comparison of AI tools within a frontend development context, thereby offering insights from within applied software engineering practice.

3 AI TRANSFORMATIVE ROLE IN SOFTWARE DEVELOPMENT

With recent progress in machine learning, natural language processing (NLP), and large language models (LLMs), AI technology has become increasingly integrated into modern development environments. Its role is not limited to improving productivity; it is also influencing how development teams collaborate and how software quality is evaluated. AI is integrated in several stages of the software development lifecycle, assisting with tasks related to coding, testing, design, and maintenance.

One of the areas where this influence can be observed is code generation and auto-completion. AI-supported tools are able to provide context-aware suggestions while developers write code. These suggestions can reduce the amount of repetitive work and, in some cases, speed up implementation, since they allow developers to focus on application logic. ML models contribute to this process by identifying common coding patterns, proposing small improvements, and pointing out possible mistakes. Although these suggestions are not always correct, they can still serve as a helpful starting point.

Machine learning techniques are increasingly applied in software testing activities. Several ML approaches focus on predicting which components or areas of a codebase are more likely to contain defects, enabling teams to focus testing efforts more effectively. This is particularly valuable in large-scale projects in order to achieve as vast test coverage as possible. By identifying high-risk modules, predictive models can enable more efficient allocation of testing resources and improve overall quality assurance processes.

In addition to predictive testing, AI-based tools are widely used to support debugging and test generation, identifying irregular patterns in code, suggesting potential fixes, and automatically generating preliminary test cases. By highlighting anomalies and suspicious code structures, these tools assist developers in detecting issues earlier in the development lifecycle. Early detection is generally associated with reduced correction costs and shorter feedback cycles; however, the overall effectiveness of these AI-driven solutions depends significantly on the quality of their outputs and the extent of human oversight during review and refinement.

Natural Language Processing has introduced new possibilities in communication and documentation. Tools such as OpenAI Codex and GPT-4 can convert natural language descriptions into code examples. This is often useful during early prototyping, when ideas are still being explored. In addition, NLP-based tools can produce summaries or explanations of existing code. Such features may support knowledge sharing within teams and make it easier for new members to become familiar with a codebase.

At a broader level, AI can provide some support in software design and architecture. For example, certain tools attempt to recommend design patterns or highlight parts of the code that could be refactored. Others can simulate aspects of system behavior to inform design choices. Reinforcement learning (RL) has also been studied in the context of testing, where models adjust test scenarios based on prior results in order to explore edge cases.

AI tools are also present in deployment and operational practices. AI-supported DevOps tools can help automate parts of continuous integration and deployment pipelines. Some systems monitor performance metrics and attempt to detect unusual behavior in production environments. In certain cases, predictive analytics are used to estimate the likelihood of failures so that preventative actions can be taken. While these approaches show promise, their success depends heavily on data quality and proper configuration.

4 CHALLENGES OF USING AI TOOLS IN SOFTWARE DEVELOPMENT

While AI tools offer significant benefits to software development, their integration and use come with a range of challenges that developers and organizations must address to fully harness their potential.

Accuracy and reliability. AI-driven tools, especially those using LLMs, can occasionally generate inaccurate or incomplete code, which can lead to bugs, security vulnerabilities, or inefficient implementations. Thus, AI-generated suggestions need to be carefully reviewed so they align with best practices and project-specific requirements.

Over-reliance on AI. Excessive dependence on AI for tasks like code generation, debugging, and documentation may reduce developers' problem-solving skills and deep technical understanding. Over time, this could lead to a decline in critical thinking, where engineers accept AI suggestions without fully evaluating their correctness or efficiency. Maintaining a balance between AI assistance and human expertise is crucial.

Ethical and security risks. AI tools may inadvertently introduce security vulnerabilities or use code that violates intellectual property rights. For instance, AI models can generate code that resembles open-source or proprietary codebases, leading to copyright issues. There is also the risk of embedding biases into the AI's decision-making, which could negatively affect the software's functionality and inclusivity.

Data privacy concerns. AI models trained on large datasets often require access to substantial amounts of code or proprietary information. This raises concerns about data privacy, especially in industries dealing with sensitive or regulated data. Unauthorized data exposure or improper model training could lead to legal and reputational risks.

Bias and inclusivity**.** AI models can inherit biases from the datasets they are trained on. If these datasets include biased examples, the AI tools may maintain or amplify those biases in generated code, recommendations or algorithms. This is particularly concerning in applications related to hiring, healthcare, and finance, where fairness and impartiality are crucial.

Integration and compatibility**.** While AI tools are powerful, integrating them into existing development workflows, systems, and tools can be a complex process. Compatibility with legacy systems or specific development environments can be an issue,

requiring additional time, effort, and resources to adapt.

Learning curve. Many AI-powered development tools come with a steep learning curve. Developers may struggle to understand how to effectively use these tools, which could reduce their overall impact. Additionally, tools that require prompt engineering or deep customization in order to work effectively, can be challenging for new users.

Lack of human context. AI tools are trained on vast datasets, but may lack the contextual understanding of a project that a human developer brings. This can lead to suggestions and code generation that may technically work but are not the most efficient solutions for the problem at hand.

5 COMPARATIVE ANALYSIS OF AI TOOLS

This paper compares three popular AI tools: Cursor, GitHub Copilot, and Gemini. We chose these three as representative examples for our empirical comparative analysis since they represent a spectrum of AI-assisted coding tools, from widely adopted solutions to emerging, innovative systems. Their differing architectures, interaction paradigms, and levels of context awareness allow for a comprehensive evaluation of code suggestion accuracy, developer productivity, and workflow integration. By including both established and cutting-edge tools, the study provides meaningful insights into the strengths and limitations of current AI coding assistants, based on their integration, learning curve, model flexibility, effectiveness in code development and applicability across different stages of development, with a primary focus on frontend development.

5.1 Integration Ease

Cursor, GitHub Copilot, and Gemini each offer different levels of integration with development tools, particularly benefiting frontend workflows.

Cursor functions as an AI-enhanced code editor based on VS Code, providing seamless compatibility with a wide range of extensions such as linters (ESLint), formatters (Prettier), and frontend frameworks like React and Vue, while also supporting native Git and GitHub integration.

GitHub Copilot offers deep integration with popular editors like VS Code and JetBrains IDEs, providing real-time code completion, frontend scaffolding, and CSS assistance, and now extends to GitHub PR workflows with Copilot Chat for interactive code reviews.

Gemini primarily integrates via APIs, Google Cloud services, and certain IDE extensions, making it especially useful for large-scale code analysis, optimization, and documentation in frontend projects, though it is less directly embedded into coding environments compared to Cursor or Copilot.

5.2 Learning Curve

The learning curve varies across Cursor, GitHub Copilot, and Gemini, depending on how naturally they fit into existing development workflows.

Cursor has a low to moderate learning curve for developers already familiar with VS Code, since it is built on top of it. Most standard coding habits transfer directly, but exploring its multi-model management and prompt engineering features can require some initial learning, especially to fully take advantage of different LLMs and custom API setups.

GitHub Copilot has a very low learning curve. It embeds directly into the typing flow of IDEs, such as: VS Code or JetBrains, offering inline suggestions and code completions. Frontend developers can start benefiting from it almost immediately, with minimal adjustment to their coding habits. Copilot Chat (for interactive conversations) may require a bit more exploration but remains intuitive.

Gemini tends to have a moderate learning curve, mainly because it is not as tightly embedded into coding IDEs. Developers often interact with Gemini via APIs, web UIs, or cloud environments (like Vertex AI Studio). To fully leverage its capabilities (large code analysis, summarization, architecture suggestions) users need to learn how to properly structure prompts and sometimes adjust their workflow to accommodate out-of-IDE interactions.

In summary, GitHub Copilot is the easiest to adopt quickly, Cursor is user-friendly for VS Code users, but requires some effort to master its multi-model capabilities, Gemini demands more adjustment due to its external/cloud-based integration.

5.3 Code Generation and Quality

The impact on code development and quality varies across Cursor, GitHub Copilot, and Gemini, depending on how each tool supports coding workflows and promotes best practices.

Cursor significantly enhances code development by supplying intelligent code completions, refactor suggestions, and in-editor AI chat assistance. Thanks to its flexible access to top-performing LLMs like

GPT-4o, Claude-3, and Gemini 2.0 Flash, Cursor generates high-quality, context-aware code, helps structure frontend components more cleanly, and encourages better coding practices. It also assists with documentation, code reviews, even generates tests, improving overall project quality.

GitHub Copilot accelerates code development speed dramatically by suggesting real-time code completions, snippets, and even complex frontend structures (React hooks, TailwindCSS layouts, Vue composables). While Copilot improves developer productivity, the quality of its suggestions varies. It sometimes produces code that works but is not always optimized or fully aligned with best practices. Developers need to actively review and refine Copilot's outputs to maintain high code quality. Therefore, while Copilot enhances productivity, the focus is on developers to ensure optimal code quality.

Gemini contributes more at a macro level, focusing on large-scale code understanding, architectural refactoring, and documentation. It is particularly strong at identifying redundant patterns, suggesting code optimizations, and helping restructure large frontend codebases. While Gemini does not directly generate small snippets during typing, its analyses can lead to cleaner, and better-architected frontend systems.

5.4 Model Flexibility

Cursor, GitHub Copilot, and Gemini each differ in the AI models they utilize and how they expose model capabilities to developers. Table 1 presents the findings.

Cursor offers extensive multi-model support, allowing users to select from a wide range of models: o1, o1-mini, GPT-4o, Claude-3.5-Sonnet, Claude-3-Opus, Gemini 2.0 Flash Exp. Additionally, Cursor allows users to integrate custom API keys from various providers, giving developers flexibility to use their own subscriptions instead of paying for Cursor's bundled service. This approach provides an advantage in cost control and model selection, especially for developers working on frontend projects requiring specific strengths from different models (e.g., faster models for autocompletion, stronger models for architectural planning).

GitHub Copilot is primarily powered by OpenAI's Codex (derived from GPT-3) and incorporates GPT-4 models, particularly in Copilot X. These models are fine-tuned for code completion, offering excellent real-time assistance in frontend development tasks such as building React/Vue components, generating CSS, and refactoring.

Gemini, through its Gemini 1.5 and Gemini 1.5 Flash models, delivers large-context AI capabilities, supporting advanced codebase analysis, summarization, and optimization. While Gemini is less tightly integrated into coding environments compared to Cursor or Copilot, its ability to handle large and complex frontend codebases makes it highly valuable for high-level refactoring and architecture design.

6 USAGE OF AI TOOLS IN VUE FRONTEND DEVELOPMENT

To objectively evaluate the effectiveness of AI-assisted coding tools in frontend development, this paper adopts a hands-on, task-based methodology. Rather than relying solely on theoretical comparisons, each tool Cursor (version 1.2), GitHub Copilot (VS Code version 1.101, powered by GPT-4), and Gemini (web interface using Gemini 2.5 Pro available at https://gemini.google.com,) was evaluated through practical implementation within a Vue 3 project scaffolded using Vite.

The selected tasks reflect common frontend challenges that developers frequently encounter: component creation, utility function abstraction, and unit testing. These tasks help assess the tools' code quality, contextual awareness, and speed of execution in a real-world development setting.

Table 1: Comparative analysis.

Tool	Integration	Models	Learning Curve	Best For
Cursor	VS Code-based	GPT-4o, Claude 3.5, Gemini 2.0, custom APIs	Low to Moderate	Flexible, high-quality code generation, multi-model workflows, larger apps
GitHub Copilot	VS Code, JetBrains, and Neovim	Codex, GPT-4	Very Low	Fast autocomplete while coding components/utilities
Gemini	API & external tools	Gemini 1.5 / Flash	Moderate	Codebase analysis, refactoring, debugging, feature-spec generation

6.1 Create a Vue Component

Objective: Create a UserCard Vue component that accepts a user prop (name, email, and avatar) and emits an onClick event when the card is clicked. The used prompt in all tools: "I am building a Vue 3 application using the Composition API. Please create a *UserCard* Vue component that accepts a user prop, containing name, email and avatar. It should emit an *onClick* event when the card is clicked. The component should follow Vue 3 best practices." The results are summarized in Table 2.

6.2 Create and Use a Utility Function

Objective: Write a reusable JavaScript function to capitalize each word in a full name string, then use it in UserCard to format the displayed name. The used prompt in all tools: "Write a reusable JavaScript utility function called *capitalizeWords* that takes a full name string and capitalizes the first letter of each word. The function should be placed in src/utils/stringUtils.js and imported into the *UserCard* component to format the displayed name." The results are summarized in Table 3.

6.3 Write Unit Tests

Objective: Write unit tests for the *UserCard* component, verifying prop rendering, event emission, and formatted name output using Jest. The used prompt in all tools: "Write unit tests for the *UserCard* Vue 3 component using Jest. The tests should verify: 1) that the name and email props render correctly, 2) that the *onClick* event is emitted when the card is clicked, and 3) that the *capitalizeWords* utility correctly formats the displayed name." The results are summarized in Table 4.

Table 2: Results from Task 1 - creating a Vue component.

Criteria	Cursor	Copilot	Gemini
Syntax Used	Vue 3 (Composition API)	Vue 2 (Options API)	Vue 2-like syntax
Functionality	Fully functional with prop and emit logic	Functional with minor Vue 3 fixes	Functional, similar to Copilot
Context Awareness	High-created files, registered components properly	Medium - generated code in isolation	Low - unaware of integration context
Time to Generate	Fast	Medium	Slow
Comments/ Explanation	Minimal - clean, working code	Minimal	Extensive code comments
Required Edits	None	Minor syntax corrections	Structural revisions and syntax updates
Code Quality	Clean, idiomatic Vue 3 Functional	Clean, idiomatic Vue 3 Functional but outdated syntax	Verbose and not Vue 3 optimized

Table 3: Results from Task 2 - working with a utility function.

Criteria	Cursor	Copilot	Gemini
Function Output	Optimized and maintainable	Optimized and maintainable	Overly complex implementation
Code Location	src/utils/stringUtils.js - follows best folder structure	utils/stringUtils.js - placed at root level	src/utils/stringUtils.js - follows best folder structure
Code Complexity	Simple, clean, and easy to follow	Simple and readable	Verbose and overengineered
Time to Generate	Fast (~10 seconds)	Fast (~10-15 seconds)	Slow (>1 minute)
Maintainability	High - easily testable and reusable	High - clear and reusable	Low - unnecessarily detailed logic
Context Awareness	High - respected existing project structure	Medium - ignored folder hierarchy	Low - limited awareness of structure
Code Quality	5/5	4/5	2/5

Table 4: Results from Task 3 -writing unit tests.

Criteria	Cursor	Copilot	Gemini
Test Coverage	Comprehensive - props, event emit, and formatted name rendering	Comprehensive and accurate	Covered main behavior but tests were overly verbose
Tooling Awareness	High - updated package.json, included install and config instructions	Medium - assumed Jest was preconfigured	Medium - included install steps but lacked integration context
Code Simplicity	Clean and readable test code	Very concise and readable	Complex and harder to follow
Time to Generate	Fast (~10 seconds)	Fast (~10-15 seconds)	Slow (>1 minute)
Maintainability	High	High	Low
Code Quality	5/5	4/5	2/5

These three small tasks showed that simple frontend features in Vue can be built more quickly with the help of modern AI tools, and they reduce the amount of manual coding needed. However, some level of developer input and correction was still required.

Among the tested tools, Cursor showed the strongest overall performance in this evaluation. It generated accurate code, was aware of the project context, and often included useful setup suggestions. Copilot was sometimes faster in producing inline suggestions, and Gemini was helpful for explanations and understanding the code. Cursor generally provided the most complete results for the chosen tasks.

7 CONCLUSIONS

This paper focused on how AI tools like Cursor, GitHub Copilot, and Gemini can help in frontend development. The goal was to compare them in real tasks and see how useful they actually are when building a Vue application.

The results indicate that these tools can save time and reduce some repetitive coding. They are especially helpful for generating components, utility functions, and test examples. However, they do not always provide high-quality results. Sometimes the code needs fixing, adjusting, or rewriting. Because of this, developers still need to check and understand what the AI produces.

Each tool showed different strengths. Cursor performed effectively when project context mattered. Copilot was fast and convenient while typing. Gemini was more useful for explanations and bigger-picture suggestions. None of them was the best in every situation. These observations are consistent with prior research, which also reports that AI coding tools provide useful assistance but still require human oversight and validation [1], [2].

This study also showed that relying too much on AI is not a particularly good idea. AI can make mistakes, and if developers trust it without any consideration, problems can appear later, even in production environments. It works best when it is used as support, not as a replacement for thinking and coding skills.

There are also some limits to this research. Only a few tasks were tested, and only in Vue. Bigger projects or other frameworks might show different results. AI tools are also improving quickly, so their performance can change over time.

Overall, AI tools are becoming part of frontend development. They can help developers work faster, but they still need human control. The best results come when developers use AI carefully and combine it with their own knowledge. AI should be viewed as a supportive tool rather than a replacement for developers.

REFERENCES

[1] S. Sarkar, “Comparative Analysis of AI-Powered Coding Assistants for Software Engineering Practices: An Evaluation of Claude Sonnet, ChatGPT, Google Gemini, and Competing Platforms,” SSRN Electronic Journal, Sep. 2025, , doi: 10.2139/ssrn.5465115.

[2] M. S. I. Ovi, N. Anjum, T. H. Bithe, M. M. Rahman and M. S. A. Smrity, “Benchmarking ChatGPT, Codeium, and GitHub Copilot: A Comparative Study of AI-Driven Programming and Debugging Assistants,” arXiv preprint arXiv:2409.19922, 2024, , doi: 10.48550/arXiv.2409.19922.

[3] E. Holloway, C. Chu, J. Tenenbaum, S. Puranic and J. Wei, “Benchmarking Code Generation Models with BIG-Bench Lite CodeGen: An Evaluation of Instruction-Tuned Large Language Models,” arXiv preprint arXiv:2304.10778, 2023, doi: 10.48550/arXiv.2304.10778.

[4] G. Shekhar, “The Impact of AI and Automation on Software Development: A Deep Dive,” ESP International Journal of Advancements in Computational Technology (ESP-IJACT), vol. 2, no. 1, pp. 162-174, 2024.

[5] U. K. Durrani, M. Akpinar, M. F. Adak, A. T. Kabakus, M. M. Öztürk and M. Saleh, “A Decade of Progress: A Systematic Literature Review on the Integration of AI in Software Engineering Phases and Activities (2013-2023),” IEEE Access, vol. 12, pp. 171185-171204, 2024, doi: 10.1109/ACCESS.2024.3488904.

[6] R. A. Husein, H. Aburajouh and C. Catal, “Large Language Models for Code Completion: A Systematic Literature Review,” Computer Standards & Interfaces, vol. 92, p. 103917, 2025, doi: 10.1016/j.csi.2024.103917.

[7] M. Barenkamp, J. Rebstadt and O. Thomas, “Applications of AI in Classical Software Engineering,” AI Perspectives, vol. 2, no. 1, 2020, , doi: 10.1186/s42467-020-00005-4.

[8] M. Adil, M. Sharif and A. Masood, “The Role of Artificial Intelligence in Software Development: A Literature Review,” Science, Engineering and Technology Review (SETR), vol. 9, no. 11, pp. 214-226, Nov. 2024.

[9] A. Brown, S. D’Angelo, A. Murillo, C. Jaspan and C. Green, “Identifying the Factors That Influence Trust in AI Code Completion,” in Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware), Porto de Galinhas, Brazil, Jul. 2024, , doi: 10.1145/3664646.3664757.

[10] A. Semenkin, V. Bibaev, Y. Sokolov, K. Krylov, A. Kalina, A. Khannanova, D. Savenkov, D. Rovdo, I. Davidenko, K. Karnaukhov, M. Vakhrushev, M. Kostyukov, M. Podvitskii, P. Surkov, Y. Golubev, N. Povarov and T. Bryksin, “Full Line Code Completion: Bringing AI to Desktop,” arXiv preprint arXiv:2405.08704, 2024, revised Jan. 8, 2025, , doi: 10.48550/arXiv.2405.08704.

[11] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer and S. Wagner, “Software Engineering for AI-Based Systems: A Survey,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 31, no. 2, art. 37e, pp. 1-59, 2022, , doi: 10.1145/3487043.