

ORIGINAL ARTICLE

An Empirical Study on the Effectiveness of Large Language Models for Automated Software Development Tasks

Marius-Romeo Pantilică and Corina Ene

¹ PIA-TECH SRL, Ploiesti, Romania

 <https://orcid.org/0009-0006-7328-8561>

e-mail: marius.pantilica@bizsolution.ro

² Department of Business Administration, Faculty of Economic Sciences, Petroleum-Gas University of Ploiesti, 39 Bucharest Avenue, 100680 Ploiesti, Romania

 <https://orcid.org/0000-0001-7790-7813>

e-mail: ecorina@upg-ploiesti.ro



Copyright: © 2025 by the authors

Citation (APA Style):

Pantilică, M.R., & Ene, C. (2025). An Empirical Study on the Effectiveness of Large Language Models for Automated Software Development Tasks. *Economic Insights – Trends and Challenges*, 14(4), 141-152. <https://doi.org/10.51865/EITC.2025.04.09>

Abstract: *This research investigates the possibility of replacing junior and intermediate programmers with artificial intelligence (AI)-based tools in the code generation process. To inspect the capabilities of these tools, the Qwen Coder tool was used to conduct the tests. The methodology used in the study is based on 10 tests, each of which presents a certain peculiarity in relation to the level of a junior or intermediate programmer. The tests include scripts, desktop applications, and web applications. Additionally, these tests also considered requirements for innovation, database handling, image processing, specific software infrastructure, and, finally, the integration of third-party services. The performance evaluation of the applications generated by the Qwen Coder tool was based on criteria that target the functional correctness of the code, the degree of understanding of the requirement, response time, the complexity of the generated application, the number of files, and the programmer's level of satisfaction. The results obtained showed that the tool generates functional code in a fraction of the time compared to a programmer's abilities, having the capacity to understand simple or medium-level tasks. In most cases, the generation time was under 5 minutes, and the applications were executed without major interventions from the programmer. A well-defined aspect in the paper addresses the necessity for the programmer to understand the specific workflow of the tool with which they generate code. In the case of the Qwen Coder tool, it requires the integration of an auxiliary tool called GitHub for managing the repositories where the code is deposited by Qwen Coder. A second aspect highlighted in the paper refers to prompt engineering, as the quality of the generated code is directly dependent on the formulation of the prompt and the level of detail within it. The results of the study show that these AI tools successfully replace entry-level and intermediate programmers.*

Keywords: *AI code generation; Large Language Models; Software Development Automation; Developer Productivity.*

JEL Classification: *C88; L86; E24; J24.*

1. Introduction

The accelerated development and integration of artificial intelligence (AI) in most fields of activity have generated numerous pressures in the automation of processes at the software engineering level. The most popular tools currently available in the field of AI are Large Language Models (LLMs). They have the ability to generate text specific to different fields of activity. In this category, software engineering is included, for which text generation refers to code generation.

Specialized AI code generation tools allow the development of software applications starting from detailed descriptions using natural language. In this way, the effort required for writing code is reduced. Consequently, the process of developing a software project is reduced both in terms of integration time and the effort expended by the programmer. This new paradigm introduces a major change in the role of the programmer. If in the past a programmer required skills in teamwork, requirement interpretation, bug identification, and understanding code written by other colleagues, currently they need to have other skills adapted to new technologies. In the category of new skills, there is the ability to coordinate the generation process, the ability to understand the code generated by these tools, the skills to identify errors and debug them, and to intervene in project settings so that it becomes functional on certain infrastructures. So, overall, the programmer must acquire entirely different skills than those of the past.

In this context, the present paper aims to analyze the performance of these automatic code generators, and Qwen Coder was used for the tests. The study is based on an experimental approach in which 10 testing scenarios inspired by real development tasks are defined. These are appropriate for the level of a junior or intermediate programmer. The scenarios cover a wide range of applications, including console applications and graphical user interface applications, which are both desktop and web-based. Each scenario is evaluated from multiple perspectives, corresponding to execution time, number of files, solution complexity, programmer satisfaction level, functionality level, and prompt interpretation. This research is important because it contributes to understanding how these tools can be used effectively in the software development process. The results provide a practical perspective on the advantages of these technologies, as well as the implications they will bring over time.

2. Literature Review

The last few years have been marked by technological evolution through the widespread introduction of artificial intelligence tools. In this category are those services that utilize large language models (LLMs) used in software content development. These technologies have changed the paradigm of software content development, impacting on the quality of the generated code, productivity, efficiency, and reducing the stress levels of content developers. The direct consequence on the labor market is visible in the reduction of the need for programmers, testers, and content developers. In the specialized literature, there is a substantial volume of materials that address various scenarios related to the use of these tools in software development (Diniță, Rosca, Stancu, & Popescu, 2025). A primary idea that emerges from these materials concerns the increase in productivity. The widespread introduction of these tools had the main objective of increasing employee efficiency at work and eliminating repetitive tasks. Studies have shown that the use of these tools in development processes, through the application of various techniques, has increased productivity by up to 55% (Ramachandran & Fouracre, 2025). From an economic perspective, this productivity translates into cost reductions and faster development of software products (Rosca & Stancu, 2025d). Cruz et al. (2025) report that this gain cannot be considered uniformly distributed, being directly dependent on the quality of the data, the application context, the type of AI tool, the skills of the person formulating the prompt,

and other elements that do not directly depend on the context (Diniță, Rosca, Tănase, & Stancu, 2025). Cruz et al. (2025) show that these benefits are limited by the complexity of tasks and the ambiguity of specifications, especially when content development relates to the generation of software systems. Studies have also shown that a large portion of developers use these tools for code generation as an industrial-level structural change (Mateo Sanguino, 2024). The integration of AI tools into IDEs increases employee efficiency on the one hand. Taking into account the axis of development at the commercial level, the implications regarding the additional costs arising from the licenses required for these tools must also be considered (Ene, 2020; Iacovoioiu & Stancu, 2017; Sergeyuk et al., 2026).

A second objective was the quality of the generated code. AI tools can generate different code for the same prompt. The generated code is mostly correct syntactically and functionally. Almanasra and Suwais (2025) show systematic deficiencies in documentation, exception handling, internal structure, integration requirement specifics, and other aspects that do not directly pertain to the quality of the code generated by the AI tool. The analysis of 600 algorithmic problems showed differences between the Java and Python languages. For the Java language, the best runtime performance was achieved for medium and difficult problems, while Python was the most memory-efficient. The results obtained by Almanasra and Suwais (2025) indicate that AI tools do not simultaneously optimize all performance dimensions. Quality issues become stringent in the context of non-functional attributes (Matei et al., 2008; Rosca, 2025).

Sun et al. (2026) present specific issues regarding the quality of LLM generators through the analysis of 109 works. The paper mentions an imbalance between academic and industrial approaches. The academic environment predominantly focuses on issues of security and performance. The industrial environment prioritizes maintainability and readability (Popescu, Gabor, & Stancu, 2024). These mentions have implications for software systems in the way evolution relates to the accumulation of technical debt. Practitioners warn that the use of AI accelerates this accumulation, resulting in a hidden cost of increased productivity (Tenea et al., 2020).

Another way to evaluate quality is the modularity of the code. This is often neglected by LLM models. Kravchuk-Kirilyuk, Graciolli and Amin (2025) show that these models generate complex and poorly structured solutions, making them difficult to manage in large systems. The lack of modularity reduces the possibility of code reuse, which translates into increased maintenance costs, counterbalancing the initial speed benefits. Thus, without the integration of software engineering principles (e.g., design-by-contract), the performance of the models remains limited in real-world contexts (Newcomb, Newcomb, & Ochoa, 2025; Voica & Stancu, 2021).

A third area of use is the security of the generated code. The analysis of 7,703 AI-generated files identified 4,241 vulnerabilities from 77 distinct types. Although 87.9% of the files do not contain detectable vulnerabilities, the vulnerability rate generates concern. The Python language exhibits rates between 16.18% and 18.50%. The JavaScript language was identified with 8.66%–8.99% and 2.50%–7.14% for TypeScript (Schreiber & Tippe, 2025; Stancu, 2015a). AI-generated code contains, on average, 6.4% more vulnerabilities than human-written code. The differences are even more pronounced in areas such as network security, where the rate increases by 18.8%, file operations by 12.4%, and error handling by 12.4%. Classic detection tools identify only about 46.7% of AI vulnerabilities, missing 53.3% of them (Yoo & Kim, 2025), which is a systemic detection issue (Dinu et al., 2018; Rosca & Stancu, 2025a).

On a larger scale, the analysis of the FormAI dataset indicates that between 51.24% and 62.07% of the generated programs are vulnerable (Tihanyi et al., 2025). These results are also confirmed by the systematic literature, which shows that common vulnerabilities include elements from the

CWE top list and are exploited in real-world scenarios (Negri-Ribalta et al., 2024; Stancu, 2012).

Concurrently with the development of code generation tools, there are advances in the development of detection tools. For example, the DeVAIC system achieves an accuracy of 94% and an average processing time of 0.14 seconds per code fragment (Cotroneo, De Luca, & Liguori, 2025). These figures suggest that automated solutions integrated into development pipelines partially compensate for the risks.

The use of generative AI models was examined by Rosca and Stancu (2026b) in the context of SQL code generation, where, based on 700 queries, differences in syntactic accuracy and response consistency were highlighted, depending on the complexity of the requirements. The paper by Rosca, Stancu and Popescu (2025) trained a machine learning (ML) model using a dataset of 90,000 queries. The paper achieved an accuracy of 96.86% in detecting SQL injection attacks.

Another determining factor for code quality is the quality of the training data. Studies show that using qualitative datasets produces better results than using raw data on a large scale. In the context of applications in fields such as health, data-driven models reduce code duplication and the need for rework, while large-scale data models exacerbate these issues (Ene, 2011; Ramachandran & Fouracre, 2025; Stancu, 2018). Thus, a productivity increase of up to 55% is achieved. In this field of medical care, the paper by Rosca, Bold and Gereia (2025) demonstrated the capabilities of the ChatGPT tool for patient classification and estimating wait times based on the level of urgency (Stancu et al., 2020).

The use of AI also has implications for developers' behavior. Studies show that there are two dominant interaction models: complete delegation of tasks to AI and using AI as consultative support. Interestingly, the second approach is correlated with higher code quality (Cruz et al., 2025; Rosca, Stancu, & Tănase, 2025). Research shows that AI tools are useful when used complementarily, not substitutively. Developers exhibit distrust in these AI tools, which is why they verify the generated code similarly to how they would check code written by other programmers (Fernandez & Cornell, 2024). At the same time, there is a tendency for overconfidence and dependency that leads to a reduction in cognitive engagement. This behavior leads to subtle errors. This duality is marked by a transition to a work model in which programmers become validators of the code. From the perspective of the labour market, the integration of AI is modifying the role of programmers by replacing coding activities with prompt engineering, response interpretation, validation, redesign, and system design activities (Carneiro Oliveira, Keuning, & Jeuring, 2025; Ene & Matei, 2012; Rosca & Stancu, 2026a). This change involves the need for continuous upskilling and the development of interdisciplinary competencies. There are also concerns about the loss of fundamental skills, especially among beginner programmers (Klemmer et al., 2024). AI is used in multiple fields, such as multilingual sentiment analysis, where a global accuracy of 81.8% was achieved in text classification (Rosca & Ariciu, 2023), preventing fraud tests on online tests based on user face and word recognition (Rosca, 2023), detecting the anomalies during health monitoring of elderly (Rosca & Stancu, 2025b), diagnosis of neurological diseases using models like SVM and neural networks (Rosca & Stancu, 2025e), monitoring soil health and detecting contaminants (Rosca & Stancu, 2025c), as well as urban traffic optimization, where AI-based systems achieved 100% accuracy in detecting road incidents (Rosca, Stancu, & Gortoescu, 2025). Another effect of integrating AI tools in coding is the increase in work complexity. It is expected that AI will reduce the time spent on coding, but the effect is to introduce additional tasks related to workload management, verification, integration, and security (Sergeyuk et al., 2026). Thus, productivity gains are partially offset by quality control costs (Ene, 2013; Stancu, 2015b). These values indicate that the economic benefits are real, but they come with technical and organizational costs.

3. Methodology

The present paper aims to conduct an analysis regarding the possibility of replacing junior programmers with AI code generation tools. To identify such a possibility, the Coder tool, owned by Qwen, will be integrated into the process of generating code, using different tasks. To evaluate the possibility of replacing junior programmers, requirements with a difficulty level appropriate to them will be used.

In this study, a set of 10 tests will be used. The 10 prompts are entered identically for both tools. Each test contains requirements that involve generating .NET code, with a level of complexity specific to junior programmer tasks. The 10 tests conducted are: generating a project that involves a meal plan, generating an e-learning platform, generating a project for text identification in images, designing and implementing a platform for CRUD operations with a SQL Server database, generating a platform for managing employees of an enterprise, designing and implementing an application for detecting AI-generated text, generating a console application for monitoring processes at the system level, generating a console application for importing data from another platform, and creating an application for counting files located in a specific location.

For implementation, the prompts were formulated to include detailed descriptions of the .NET Framework version used, the programming language, the application structure, the program inputs, and the expected outcome. The use of Coder requires a convoluted setup because it necessitates integration with a GitHub account and the creation of a repository to store the generated code. Subsequently, the code was manually downloaded and run locally for validation. This workflow introduced additional steps in the experimental process, which were taken into account in the comparative analysis. To illustrate the workflow with the Coder instrument, Figure 1 presents it.

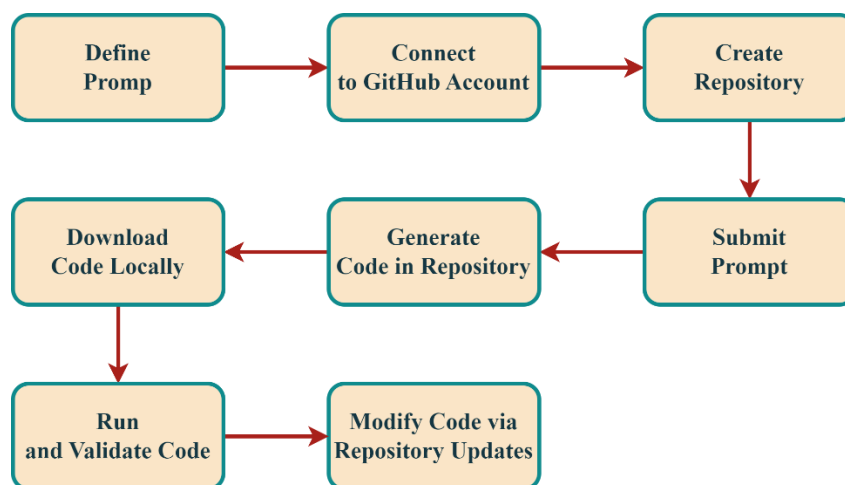


Fig. 1. Logical Flow for the Coder Tools

For the validation of the results provided by the Coder tool, the first evaluation criterion used was the functionality of the code. Subsequently, the complexity of using the tool, response time, prompt formulation complexity, and flexibility in making subsequent modifications were analyzed. Table 1 presents the 10 prompts evaluated for the Coder tool.

Table 1. The 10 Prompts Evaluated for The Coder Tool

Test	Prompt
Test 1 – Meal Plan Application	Generate a .NET Framework application in C# that implements a meal planning system. The application should allow users to create, edit, and delete meals, assign meals to specific days of the week, and display a weekly meal plan. Include basic data validation and use an appropriate data structure to store meals.
Test 2 – E-learning Platform	I need an ASP.NET Core project that has a user module (registration, authentication, password reset, email confirmation). There are 3 types of users: administrator, student, and teacher. Teachers set up courses on the platform, and each course is organized into modules. Each module has study materials (documents, video tutorials, and knowledge assessment tests). Each completed module is associated with a score, and based on the score and the time spent by the student per accessed module, it must be monitored so that the modules are listed according to interests and this time based on a unique formula for optimizing module visualization. The database is SQL Server. I need you to also provide me with the queries for the database structure.
Test 3 – Image Text Identification	Generate a .NET Framework application in C# that extracts and identifies text from images. Use an OCR approach (e.g., Tesseract or a similar library). The application should allow users to upload an image and display the extracted text. Use WPF Template.
Test 4 – CRUD Application with SQL Server	Generate a .NET Framework application in C# that implements CRUD (Create, Read, Update, Delete) operations using a SQL Server database. Include a sample table (e.g., Products or Students) and demonstrate database connection, queries, and basic error handling.
Test 5 – Employee Management System	Generate a .NET Framework application in C# for managing employees in an enterprise. The system should allow adding, updating, deleting, and viewing employee records, including fields such as name, position, salary, and department.
Test 6 – AI-generated Text Detection	Generate a .NET Framework application in C# that detects whether a given text is AI-generated or human-written. Implement simple heuristics or integrate an external API if necessary. The application should allow input text and display a classification result.
Test 7 – System Process Monitoring	Generate a .NET Framework console application in C# that monitors system processes. The application should list running processes, display CPU and memory usage, and allow filtering by process name.
Test 8 – Data Import Application	Generate a .NET Framework console application in C# that imports data from an external platform (e.g., CSV file or API). The application should read the data, parse it, and store it in a structured format.
Test 9 – File Counting Application	Generate a .NET Framework application in C# that counts the number of files in a specified directory. The application should allow the user to input a file path and display the total number of files, optionally grouped by file type.
Test 10 – Meal Plan	I want a C# console application .NET Framework 4.7.2 for a meal plan based on the inputs: height (cm), weight (kg), age (years), sex (M/F), physical activity (Intense/Moderate/Sedentary). At the output, I need to have the caloric requirement to be in deficit. It would also be useful to have a dietary plan suggestion, considering that I will use the OpenAI key for this.

The evaluation of the results was based on several criteria: the functional correctness of the code (FCF), the degree of understanding of the prompt (DUP), the response time (RT), the application complexity (AC), the number of generated files (NF), and the programmer's satisfaction level (PSL).

4. Results

After conducting the 10 tests, the results obtained were synthesized in relation to the performance criteria established in the methodology. The interpretation of these criteria forms the picture regarding the capacity of the Queen Coder tool in automatic code generation. In this way, it will be possible to evaluate the potential of replacing junior-level programmers with such tools capable of solving low to medium-level difficulty tasks. The 10 tests are evaluated against the criteria FCF, DUP, RT, AC, NF, and PSL. To synthesize the results, Table 2 was constructed, in which medium, high, or low labels were assigned based on performance: high if it performed very well, or low if it had poor results.

Table 2. Results Summary Table

Test	FCF	DUP	RT (min: sec)	AC	NF	PSL
Test 1	High	Medium	1:50	Low	2	Medium
Test 2	Low	High	10:54	High	13	Medium
Test 3	High	High	4:38	Medium	4	High
Test 4	High	High	2:53	Medium	8	High
Test 5	High	High	2:18	Medium	7	High
Test 6	Medium	Low	3:13	Low	3	Low
Test 7	High	High	1:26	Low	2	High
Test 8	None	Low	N/A	Medium	N/A	Very Low
Test 9	High	High	1:25	Low	2	High
Test 10	High	High	2:28	Medium	4	High

Following the tests for the first prompt, the tool's ability to generate code for a console application was taken into consideration. In the case of this test, the execution time was 1m50s. The code was 100% functional, but it failed to understand the requirement correctly. In the case of this prompt, the Coder generated a project in which the application user sets up their own meals and uses the application as a planner journal to follow. The application is a console type, so the level of complexity is low. The number of files generated was 2.

The second prompt aimed at generating a complex project, which involves the existence of a database, working with access restrictions, thus a large volume of work, with an intermediate level of difficulty. The execution time was 10m54s. The code is not functional. The application is of the ASP.NET Core type, and Coder generated the entire underlying structure. The tool generated 13 files that would have taken at least 20 days of work to write.

The third prompt aimed to generate an application that focuses on the research area, specifically for image processing. The project was generated in 4m38s. The graphical interface was generated entirely from C# code. The tool installed Tesseract for character extraction from images. The application is functional; it fulfills its purpose for a limited set of tests conducted, and the only modification made was the selection of the architecture it runs on, namely x64. The project contains 4 files.

Prompt four aims to integrate the minimal functionalities specific to a database. This type of application is specific to junior programmers and is most often used for their evaluation by hiring companies. The application is 100% functional; it was generated in 2m53s, it generated the database structure, created classes, handled possible error messages, so it adhered to all the best practices in coding. Eight files were generated.

The fifth prompt corresponds to the work with the database integrated into a console-type application. The generated project is 100% functional and meets all the tasks mentioned in the requirements. It contains 7 files, which were generated in 2m18s. The application does not use a database because the prompt did not include the use of a database in its description.

The sixth prompt is based on an innovation criterion. The generated project is 100% functional, but it does not meet its objective. In other words, the logic generated at the code level does not meet the tasks outlined in the requirement. The project contains 3 files, generated in 3m13s.

The seventh prompt aims to inspect the capabilities of the Qwen Coder tool to work on operating system-dependent configurations. The generated project is 100% functional, contains 2 files that were generated in 1m26s, and meets all the requirements from the prompt.

The eighth prompt aims to generate a massive data import script. For this prompt, the upload to GitHub could not be completed. I set up 3 different repositories, but none of them could complete the deployment. The message received was "The branch update was rejected by the remote server." Please verify the repository's status and ensure your GitHub credentials are correct and up to date," without there actually being an error in the account settings.

The penultimate prompt aimed to inspect the possibility of generating code that works with the file and directory system. For this prompt, 2 files were generated in 1m25s. The code is 100% functional and meets the requirements in full.

The last prompt aimed to generate a complex application, with an explicit, detailed description, to minimize the degree of uncertainty for the Qwen Coder tool. The latter generated 4 files in 2m28s. The code is 100% functional, and it meets all the requirements. I had to set the OpenAI key. The code is very well structured and easy to understand. It is worth noting that Qwen Coder supports the execution of a maximum of 2 tasks in parallel.

5. Discussions

From the analysis of the results, it is noted that Qwen Coder exhibits performance typical of an intermediate junior programmer. It is noted that the tool encounters difficulties in handling ambiguous requirements. From the perspective of functional correctness, seven out of the ten applications generated fully functional projects. The 30% of cases where the projects were not generated functionally stem from the prompts not being detailed enough. The 70% success rate in code generation highlights the fact that the tool produces executable code without major interventions, which revolutionizes the field of programming. A detailed analysis of the generated projects indicates that technical functionality differs from conceptual correctness. A concrete example can be analyzed in the case of test 6, where the generated application was completely functional from a technical standpoint, but did not meet the required objective. This aspect highlights that a tool can generate code correctly when the technology is known or the user details the technology that needs to be implemented at the code level. A second idea that can be described from this test comes from the fact that the tool could perform much better if it were integrated into a workflow that interconnects multiple AI services with different capabilities. A third observation is related to the quality of the prompt. Generic descriptions create confusion for the code-generating tool. Regarding the response time, the obtained values range between 1 minute and 11 minutes. The average execution time is approximately 3 minutes and 40 seconds. This value is considered by the author of this work to be exceptional in the context of automated code generation. From the values, a correlation between the application's complexity and the response time is also observed. Thus, console-type applications were generated in approximately 1-2 minutes, while graphical interface applications required between 3 and 5 minutes. Test 2 is a particular case, as the complexity of the requirement led to a much longer execution time. Another important aspect is the complexity of the generated applications. Most applications fall into the medium complexity category. These are structured on several files that range between 2 and 12, with an average of approximately 4-5 files per application. Test 2 generated 13 files due to the increased complexity associated with the project.

The programmer's level of satisfaction is correlated with the correctness of the generated code. In six out of the ten tests conducted, the programmer experienced a state of satisfaction because

the code was perfectly functional. This aspect demonstrates a positive experience in the majority of cases. For test 8, where the deployment to GitHub was unsuccessful, the programmer's satisfaction was affected. The result highlights a limitation of the tool regarding the correct generation of code.

Compared to other tools, a disadvantage of the Qwen Coder tool is the need to use the second tool, GitHub, as an auxiliary for code sharing. In the deployment process, these additional steps introduce frustration for the programmer, who feels that they need to make extra settings that might have been omitted.

Regarding the type of applications, the results obtained with the help of the tool demonstrate that utility applications or those based on detailed logic, such as process monitoring or file counting, perform better than the other tests. In these cases, the execution time is very good. An interesting result obtained from the tests conducted is that where the prompt was vaguely formulated, the application's structure was affected. For example, in test 5, the absence of explicit requirements regarding database usage resulted in applications without data persistence. The obtained results demonstrate that the Qwen Coder tool is effective in generating code associated with applications of a medium difficulty level. The results are functional in most prompts, and the time taken to generate them is considered negligible compared to the effort put in by a programmer to generate the same code.

6. Conclusions

In this research, the aim was to evaluate the impact of specific AI tools in code generation. Additionally, the performance of the Qwen Coder tool was analyzed, and the results showed the possibility of replacing junior and intermediate programmers with these new software development technologies.

The key findings of the research are:

- Strong performance in simple tasks with a high success rate over 90% for low-complexity tasks;
- Decreasing reliability with complexity, as functional failure occurs in a high-complexity scenario;
- Semantic understanding limitations generated by the misinterpretation in conceptual tasks;
- Workflow constraints, by being dependent on GitHub, introduce friction;
- Scalability issues generated by the parallel task limitation (max 2 tasks).

One initial aspect that can be inferred from conducting these tests pertains to the reduction in the time required for application development. In the experiment, most applications were generated in specific minute intervals. In other words, the majority of projects required between 1 and 4 minutes, which is far superior to human capabilities. A detailed analysis at the test level reveals that these tools are high-performing. In the case of simple applications, the results provided 100% functional code that correctly interpreted the programmer's requests. The tests included scripts, desktop applications with graphical interfaces, and web applications. In the case of complex web platform applications, a series of discrepancies were observed between the generated structure and the requested functionality. However, the author of this paper acknowledges the exceptional ability of the tool in minimizing the time and effort expended by the programmer in developing such a web platform.

Also, within this study, a difference was observed between technical accuracy and conceptual accuracy. The author of this paper emphasizes that this difference arises from the quality of the provided prompt. Future research will focus exclusively on the formulation of these prompts in relation to the correct generation of code that implements the project. Also in these tests, the

ability of these tools to replace junior and intermediate-level programmers was noted. On the other hand, programmers who use this new technology require knowledge of the workflow, meticulousness in analyzing the generated code, as well as skills in debugging the code produced by these tools, and flexibility in working with code that was not written by the respective programmer.

In conclusion, the research ends with the general conclusion regarding the technological advancement generated by the introduction of these tools capable of automatically writing code using developments in the field of software engineering.

References

1. Almanasra, S., & Suwais, K. (2025). Analysis of ChatGPT-Generated Codes Across Multiple Programming Languages. *IEEE Access*, 13, 23580-23596. <https://doi.org/10.1109/access.2025.3538050>
2. Carneiro Oliveira, E., Keuning, H., & Jeurig, J. (2025, 11-16 November). Uncovering Behavioral Patterns in Student-LLM Conversations during Code Refactoring Tasks. The 25th Koli Calling International Conference on Computing Education Research, Koli, Finland. <https://doi.org/10.1145/3769994.3770000>
3. Cotroneo, D., De Luca, R., & Liguori, P. (2025). DeVAIC: A tool for security assessment of AI-generated code. *Information and Software Technology*, 177, 107572. <https://doi.org/10.1016/j.infsof.2024.107572>
4. Cruz, E. D. L., Le, H., Meduri, K., Nadella, G. S., & Gonaygunta, H. (2025). Redefining the Programmer: Human-AI Collaboration, LLMs, and Security in Modern Software Engineering. *Computers, Materials & Continua*, 85(2), 3569-3582. <https://doi.org/10.32604/cmc.2025.068137>
5. Diniță, A., Rosca, C.-M., Stancu, A., & Popescu, C. (2025). Distributed IoT-Based Predictive Maintenance Framework for Solar Panels Using Cloud Machine Learning in Industry 4.0. *Sustainability*, 17(21), 9412. <https://doi.org/10.3390/su17219412>
6. Diniță, A., Rosca, C.-M., Tănase, M., & Stancu, A. (2025). A Comprehensive Review on Bridging the Research Gap in AI-Driven Material Simulation for FRP Composites. *Computer Modeling in Engineering & Sciences*, 144(1), 147-199. <https://doi.org/10.32604/cmes.2025.066276>
7. Dinu, C., Ungureanu, E. M., Vasile, G. G., Kim, L., Ionescu, I., Ene, C., & Simion, M. (2018). Soil and Vegetation Pollution from an Abandoned Mining Area Situated in Hunedoara County, Romania. *Revista de Chimie*, 69(1), 14-20. <https://doi.org/10.37358/RC.18.1.6036>
8. Ene, C. (2011). Rapex system - an efficient tool for european consumer safety. *The USV Annals of Economics and Public Administration*, 11(1(13)). <http://annals.seap.usv.ro/index.php/annals/article/view/367/377>
9. Ene, C. (2013). Post-Consumer Waste – Challenges, Trends, Solutions. *International Journal of Sustainable Economies Management (IJSEM)*, Volume 2, Issue 3 (July-September 2013), ISSN: 2160-9659, EISSN: 2160-9667, pp.19-31.
10. Ene, C. (2020). Food Security and Food Safety: Meanings and Connections *Economic Insights – Trends and Challenges, IX(LXXII)* (1), 59-68. https://upg-bulletin-se.ro/wp-content/uploads/2020/12/7.Ene_.pdf
11. Ene, C., & Matei, M. (2012, 1 October). New dimensions of food security in Romania from the European perspective. Agrarian Economy and Rural Development - Realities and Perspectives for Romania. 3rd Edition of the International Symposium, Bucharest. Romania. <https://www.econstor.eu/bitstream/10419/76851/1/749368993.pdf>
12. Fernandez, A. S., & Cornell, K. A. (2024, 20-23 March). CS1 with a Side of AI: Teaching Software Verification for Secure Code in the Era of Generative AI. The 55th ACM Technical Symposium on Computer Science Education, Portland, USA. <https://doi.org/10.1145/3626252.3630817>
13. Iacovoiu, V., & Stancu, A. (2017). Competition and Consumer Protection in the Romanian Banking Sector. *Amfiteatru Economic*, 19(45), 381-396. <https://www.amfiteatruconomic.ro/ArticolEN.aspx?CodArticol=2625>
14. Klemmer, J. H., Horstmann, S. A., Patnaik, N., Ludden, C., Burton, C., Powers, C., Massacci, F., Rahman, A., Votipka, D., Lipford, H. R., Rashid, A., Naiakshina, A., & Fahl, S. (2024, 14-18 October). Using AI Assistants in Software Development: A Qualitative Study on Security Practices

- and Concerns. CCS '24: ACM SIGSAC Conference on Computer and Communications Security, Salt Lake City, USA <https://doi.org/10.1145/3658644.3690283>
15. Kravchuk-Kirilyuk, A., Gracioli, F., & Amin, N. (2025, 12-18 October). The Modular Imperative: Rethinking LLMs for Maintainable Software. 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages, Singapore. <https://doi.org/10.1145/3759425.3763392>
 16. Matei, M., Stancu, A., Enescu, G., & Geambașu, C. (2008). *Burse de mărfuri și valori*. Petroleum-Gas University of Ploiesti.
 17. Mateo Sanguino, T. D. J. (2024). Enhancing Security in Industrial Application Development: Case Study on Self-Generating Artificial Intelligence Tools. *Applied Sciences*, 14(9), 3780. <https://doi.org/10.3390/app14093780>
 18. Negri-Ribalta, C., Geraud-Stewart, R., Sergeeva, A., & Lenzi, G. (2024). A systematic literature review on the impact of AI models on the security of code generation. *Frontiers in Big Data*, 7, 1386720. <https://doi.org/10.3389/fdata.2024.1386720>
 19. Newcomb, L., Newcomb, A., & Ochoa, O. (2025). Preconditions and Postconditions as Design Constraints for LLM Code Generation. *IEEE Access*, 13, 184259-184274. <https://doi.org/10.1109/access.2025.3625819>
 20. Popescu, C., Gabor, M. R., & Stancu, A. (2024). Predictors for Green Energy vs. Fossil Fuels: The Case of Industrial Waste and Biogases in European Union Context. *Agronomy*, 14(7), 1459. <https://doi.org/10.3390/agronomy14071459>
 21. Ramachandran, M., & Fouracre, S. (2025). Rich Data Versus Quantity of Data in Code Generation AI: A Paradigm Shift for Healthcare. *Blockchain in Healthcare Today*, 8(5), 396. <https://doi.org/10.30953/bhty.v8.396>
 22. Rosca, C.-M. (2023). New Algorithm to Prevent Online Test Fraud Based on Cognitive Services and Input Devices Events. In A. Noor, K. Saroha, E. Pricop, A. Sen, & G. Trivedi (Eds.), *Proceedings of Third Emerging Trends and Technologies on Intelligent Systems. ETTIS 2023. Lecture Notes in Networks and Systems* (Vol. 730, pp. 207-219). Springer Nature, Singapore. https://doi.org/10.1007/978-981-99-3963-3_17
 23. Rosca, C.-M., Bold, R.-A., & Gere, A.-E. (2025). A Comprehensive Patient Triage Algorithm Incorporating ChatGPT API for Symptom-Based Healthcare Decision-Making. *Emerging Trends and Technologies on Intelligent Systems. ETTIS 2024. Lecture Notes in Networks and Systems*, Noida, India. https://doi.org/10.1007/978-981-97-5703-9_13
 24. Rosca, C.-M., & Stancu, A. (2025a). AI Anomaly-Based Deepfake Detection Using Customized Mahalanobis Distance and Head Pose with Facial Landmarks. *Applied Sciences*, 15(17), 9574. <https://doi.org/10.3390/app15179574>
 25. Rosca, C.-M., & Stancu, A. (2025b). Anomaly Detection in Elderly Health Monitoring via IoT for Timely Interventions. *Applied Sciences*, 15(13), 7272. <https://doi.org/10.3390/app15137272>
 26. Rosca, C.-M., & Stancu, A. (2025c). Emerging Trends in AI-Based Soil Contamination Monitoring and Prevention. *Agriculture*, 15(12), 1280. <https://doi.org/10.3390/agriculture15121280>
 27. Rosca, C.-M., & Stancu, A. (2025d). Innovative AIoT Solutions for PET Waste Collection in the Circular Economy Towards a Sustainable Future. *Applied Sciences*, 15(13), 7353. <https://doi.org/10.3390/app15137353>
 28. Rosca, C.-M., & Stancu, A. (2025e). A Review of Neuro-ML Breakthroughs in Addressing Neurological Disorders. *Applied Sciences*, 15(10), 5442. <https://doi.org/10.3390/app15105442>
 29. Rosca, C.-M., & Stancu, A. (2026a). Machine Learning Applications with Sensors for Indoor Air Quality Research. *Sensors*, 26(9), 2909. <https://doi.org/10.3390/s26092909>
 30. Rosca, C.-M., & Stancu, A. (2026b). Quality assessment of GPT-3.5 and Gemini 1.0 Pro for SQL syntax. *Computer Standards & Interfaces*, 95, 104041. <https://doi.org/10.1016/j.csi.2025.104041>
 31. Rosca, C.-M., Stancu, A., & Gortoescu, I.-A. (2025). Advanced Sensor Integration and AI Architectures for Next-Generation Traffic Navigation. *Applied Sciences*, 15(8), 4301. <https://doi.org/10.3390/app15084301>
 32. Rosca, C.-M., Stancu, A., & Popescu, C. (2025). Machine Learning Models for SQL Injection Detection. *Electronics*, 14(17), 3420. <https://doi.org/10.3390/electronics14173420>
 33. Rosca, C.-M., Stancu, A., & Tănase, M. R. (2025). A Comparative Study of Azure Custom Vision Versus Google Vision API Integrated into AI Custom Models Using Object Classification for Residential Waste. *Applied Sciences*, 15(7), 3869. <https://doi.org/10.3390/app15073869>
 34. Rosca, C. M. (2025). Convergence Catalysts: Exploring the Fusion of Embedded Systems, IoT, and Artificial Intelligence. In X.-S. Yang (Ed.), *Engineering Applications of AI and Swarm Intelligence* (pp. 69-87). Springer Nature. https://doi.org/10.1007/978-981-97-5979-8_4

35. Rosca, C. M., & Ariciu, A. V. (2023). Unlocking Customer Sentiment Insights with Azure Sentiment Analysis: A Comprehensive Review and Analysis. *Romanian Journal of Petroleum & Gas Technology, IV (LXXV)*(1), 173-182. <https://doi.org/10.51865/JPGT.2023.01.15>
36. Schreiber, M., & Tippe, P. (2025, 29 October). Security Vulnerabilities in AI-Generated Code: A Large-Scale Analysis of Public GitHub Repositories. International Conference on Information and Communications Security, Nanjing, China. https://doi.org/10.1007/978-981-95-3537-8_9
37. Sergeyuk, A., Zakharov, I., Koshchenko, E., & Izadi, M. (2026). Human-AI experience in integrated development environments: a systematic literature review. *Empirical Software Engineering, 31*(3), 55. <https://doi.org/10.1007/s10664-025-10793-0>
38. Stancu, A. (2012). Food and Feed Safety in Romania in the European Union Context: Current Issues. *Economic Insights – Trends and Challenges, I(LXIV)*(3), 87-95. https://upg-bulletin-se.ro/old_site/archive/2012-3/9.Stancu.pdf
39. Stancu, A. (2015a). An analysis of the relation between wine consumption and cultural models. *Ekonomika poljoprivrede, 62*(1), 207-227. <https://doi.org/10.5937/ekopolj1501207s>
40. Stancu, A. (2015b). The Relationship among Population Number, Food Domestic Consumption and Food Consumer Expenditure for Most Populous Countries. *Procedia Economics and Finance, 22*, 333-342. [https://doi.org/10.1016/s2212-5671\(15\)00298-1](https://doi.org/10.1016/s2212-5671(15)00298-1)
41. Stancu, A. (2018). ASF Evolution and Its Economic Impact in Europe over the Past Decade. *The USV Annals of Economics and Public Administration, 18*(2(28)), 18-27. <http://www.annals.feaa.usv.ro/index.php/annals/article/view/1086/923>
42. Stancu, A., Panait, M., Raimi, L., & Palazzo, M. (2020, 15-16 October). Financial consumers—promoters of sustainable development? Evidences from Europe and Central Asia. 3rd International Conference on Economics and Social Sciences, Bucharest, Romania. <https://doi.org/10.2478/9788395815072-083>
43. Sun, X., Ståhl, D., Sandahl, K., & Kessler, C. (2026). Quality assurance of LLM-generated code: Addressing non-functional quality characteristics. *Journal of Systems and Software, 238*, 112885. <https://doi.org/10.1016/j.jss.2026.112885>
44. Tenea, A. G., Vasile, G. G., Dinu, C., Gheorghe, S., Pascu, L. F., Mureseanu, M., & Ene, C. (2020). Behavior of Cd Accumulation in *Sinapis alba* L. in the Presence of Essential Elements (Ca, Mg, Fe, Zn, Mn, Cu, Ni). *Revista de Chimie, 71*(6), 378-389. <https://doi.org/10.37358/rc.20.6.8204>
45. Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., & Cordeiro, L. C. (2025). How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering, 30*(2), 47. <https://doi.org/10.1007/s10664-024-10590-1>
46. Voica, M. C., & Stancu, A. (2021). Corporate Social Responsibility Reporting: Background, Evolution and Sustainability Promoter. In C. F. Machado & J. P. Davim (Eds.), *Sustainable Management for Managers and Engineers* (pp. 109-155). ISTE Wiley. <https://doi.org/10.1002/9781119804345.ch6>
47. Yoo, S. H., & Kim, H. J. (2025). Security Analysis of Automated Code Generation. *Tehnički glasnik, 19*(4), 560-574. <https://doi.org/10.31803/tg-20250225095135>