

Comparative Analysis of the Applicability of Artificial Intelligence Models for Code Generation

Patryk Warchoł*, Małgorzata Plechawska-Wójcik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This article analyses the potential for using selected artificial intelligence models in the code generation process during the development of a web application. The analysis was conducted in the Windsurf environment using the Open Data Explorer project as a case study. These models were compared in terms of functional effectiveness, implementation efficiency, the quality of the generated code, project stability, and the types of errors that occurred during subsequent iterations. The results of the analysis were presented in the form of tables and comparative descriptions, which allowed for an assessment of the practical usefulness of the studied models in AI-assisted programming.

Keywords: AI; LLM; code generation; AI-assisted programming; code quality; IDE

*Corresponding author

Email address: patryk.warchol2k@gmail.com (P. Warchoł)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

The rapid development of generative artificial intelligence in recent years has significantly impacted the way software development is carried out. Large language models, as well as model-based tools such as ChatGPT and GitHub Copilot, are increasingly being used not only to generate individual code snippets, but also to support debugging, error analysis, explaining program behaviour, and automating selected programming tasks.

AI-native IDEs are particularly significant. In this approach, the model does not merely serve as an external conversational tool, but directly supports the code-writing process.

The motivation for addressing this topic stems from the growing importance of AI tools in the actual software development process and the need to assess how selected models perform when implementing the various stages of a larger software development project.

The aim of this paper is to conduct a comparative analysis of the use of selected artificial intelligence models for code generation during the development of a web application in the Windsurf environment. The paper focuses on the practical application of language models that support programming in conditions similar to those of a real-world, iterative software development process.

Based on the literature review and the identified research gaps, the following hypotheses were formulated:

1. H1: The artificial intelligence models analysed differ in terms of functional effectiveness, defined as the degree to which the required web application functionalities are correctly implemented in the Windsurf environment.
2. H2: The models under analysis differ in terms of application development efficiency, defined as the number of iterations and the time required to produce a working solution.
3. H3: Greater functional efficiency of a model does not necessarily imply higher quality of the generated

code, understood as readability, modularity, maintainability, and compliance with the established project structure.

4. H4: The models under analysis differ in their ability to maintain design consistency across successive iterations, particularly in terms of preserving existing functionality and ensuring that new changes are compatible with the application's existing structure.
5. H5: The models analysed differ in the types and frequency of errors made when generating, modifying, and correcting application code.

2. Literature Review

2.1. LLM in code generation

In the context of code generation, LLMs process a task description written in natural language, a snippet of existing code, or the broader context of a project, and consequently generate an implementation proposal [1], [2]. Their applications include, among others, creating new functions, completing program fragments, supporting debugging, preparing tests, documenting code, and assisting in the process of understanding existing implementations [3]. In practice, this can reduce some of a programmer's routine tasks, such as writing repetitive code snippets, searching for common solutions, or preparing basic application structures [3].

The use of large language models represents an evolution of earlier recommendation systems and intelligent developer support tools. Earlier solutions focused mainly on syntax suggestions, searching for code examples, recommending libraries, or supporting refactoring [4]. Modern generative models expand this scope because they can generate longer implementations, explain how a program works, and respond to user feedback [1], [3], [5].

2.2. Assessment of the quality of the generated code

Assessing the quality of code generated by large language models is one of the key areas of research into the

use of artificial intelligence in software engineering. The mere fact that syntactically correct code has been generated does not necessarily mean that the solution meets functional requirements or that it is efficient, secure, and maintainable [6]. The literature emphasizes that the quality of code generated by artificial intelligence models should be analysed from multiple perspectives. This includes program correctness, complexity, error susceptibility, security, the risk of performance regressions, readability, modularity, and maintainability [6, 7, 8].

A significant challenge in evaluating code generated by language models is hallucinations and characteristic generative errors. In the context of source code, these can manifest as the use of non-existent functions, incorrect arguments, invalid attribute names, incomplete solution generation, misinterpretation of commands, lack of edge case handling, or the addition of unsolicited functional elements [9, 10]. Such problems can be difficult to detect because the code may appear correct while simultaneously leading to runtime errors or non-compliance with task requirements. This underscores the necessity of testing as well as critically analysing solutions generated by artificial intelligence models [11].

2.3. Human–AI Interaction in Programming

The development of tools that utilize LLMs has significantly changed the way a programmer's work is organized. In an AI-assisted development model, certain tasks—such as proposing implementations, explaining code, diagnosing errors, or suggesting next steps—are shifted to interactions with the language model. This does not, however, mean replacing the programmer, but rather a change in the nature of their work: from the author of each piece of code to the person directing the generation process, verifying proposals, and making design decisions [12, 13].

One of the most important aspects of this shift is the transition from one-time code generation to an iterative workflow. In practical applications, the solution generated by the model often requires clarification, verification, or correction. As a result, the workflow involves formulating an initial prompt, analysing the response, reporting errors, and gradually refining the solution. The literature on conversational interactions with ChatGPT in programming projects shows that programmers often engage in multi-turn conversations, with subsequent commands serving to clarify requirements, correct code, explain program behaviour, test, or refactor [14].

In larger projects, maintaining context is particularly important. The model should take into account the file structure, previous implementation decisions, dependencies between components, the adopted coding style, and the current state of the application [2]. A lack of full context can lead to the generation of solutions inconsistent with the project, misinterpretation of the task, incomplete implementation, or the addition of unsolicited functional elements [2], [10]. For this reason, the

effective use of language models requires both properly conducted interaction and access to project information.

2.4. Research gap

The reviewed literature indicates that a significant portion of research on the use of large language models in code generation focuses on individual tasks, short code snippets, algorithmic benchmarks, or selected quality aspects, such as functional correctness, security, performance, or complexity [7, 8]. While this approach allows for systematic model comparison, it only partially reflects the actual software development process, in which an application is built in stages, requires maintaining project context, integrating multiple components, and repeatedly refining the code within the developer's work environment [6].

The literature reviewed thus reveals a gap regarding the practical evaluation of language models in the process of developing a complete application, rather than merely solving isolated programming tasks. It is also important to consider the iterative nature of working with a model, which includes formulating commands, analysing responses, reporting errors, correcting implementations, and developing new functionalities [14], [5]. These identified gaps justify conducting a research experiment focused on the practical use of artificial intelligence models in the process of developing a web application in the Windsurf environment.

3. Research Methodology and Experiment Description

3.1. Research subject: Open Data Explorer

Open Data Explorer is a single-page web application with a controlled yet diverse range of features, where all key functions are accessible through a single user interface. The application allows users to:

- import data from CSV, XLS, and XLSX files;
- validate the data;
- analyse the data in terms of types, structure, and missing values;
- configure and generate visualisations;
- prepare a PDF report summarizing the key information about the dataset.

From a research perspective, it is significant that the project encompasses several different classes of programming tasks. This makes it possible to evaluate artificial intelligence models not only in the context of individual responses to simple commands, but also in the process of building a complete system developed in stages.

The decision to choose the Open Data Explorer application as a research project was based on both methodological and practical considerations. The project has a moderate level of complexity. It is sufficiently comprehensive to highlight differences between models, yet limited enough to allow for its full implementation within a controlled experiment.

3.2. Development environment and analysed models

The main tool used in the experiment was Windsurf [15], an integrated development environment supported by artificial intelligence models. The choice of Windsurf as the primary research tool was based on its suitability for the work scenario, in which the model does not merely answer individual questions but participates in the gradual development of the application.

In this IDE, the models operated directly within the context of the project's current state. A key feature of the environment was the ability to reference existing files, previous code snippets, and the application's structure. From the perspective of the experiment, this was particularly significant because it allowed for an analysis of the effectiveness of code generation and the models' ability to maintain project consistency throughout subsequent implementation stages.

For the purposes of the experiment, a simplified application architecture was adopted, tailored to the iterative nature of the software development process using language models and to the objective of the study, which is to evaluate the models' ability to build a working solution. For this reason, the classic division into separate client and server components was abandoned, and a unified application based on the Python language was chosen. The logical structure of the Open Data Explorer application and the dependencies between the system's main modules are shown in Figure 1.

Using the Streamlit framework allows to combine the user interface layer and application logic into a single environment. This approach is well suited to analytical applications, as it enables the rapid creation of interactive interfaces, the display of tables, the handling of input files, and the generation of charts.

Table 2 provides a description of the development environment.

The experiment utilized three artificial intelligence models (Table 1) representing different approaches to code generation.

Table 1: AI models analysed in the experiment

Publisher	Model	Version
OpenAI	ChatGPT	5.5
Anthropic	Claude	Opus 4.7
Codeium	SWE	1.6

Table 2: Experimental environment configuration

Environment component	Version
Operating system	Windows 11
Python interpreter	3.12.10
Isolated environment	.venv
Application framework	Streamlit 1.57.0
Data libraries	Pandas 3.0.3, NumPy 2.4.6
Visualization libraries	Matplotlib 3.10.0, plotly 6.7.0, seaborn 0.13.2
PDF libraries	ReportLab 4.5.1
IDE	Windsurf 2.3.9

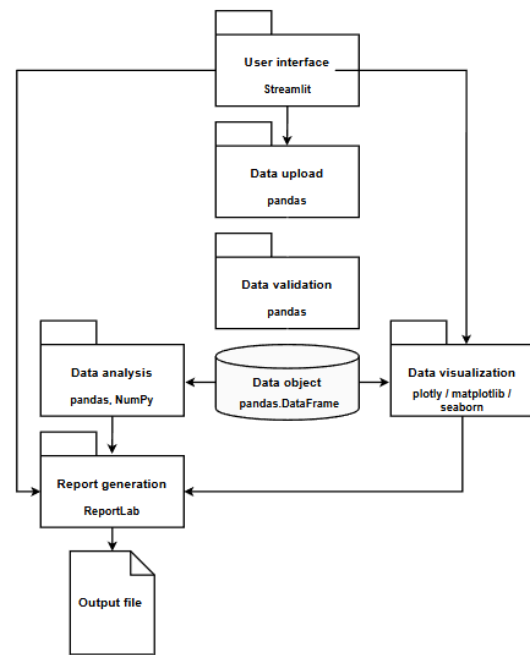


Figure 1: Simplified package diagram of the Open Data Explorer application

3.3. Test Procedure

The study was designed as a controlled comparative experiment aimed at evaluating the performance of language models used to support the iterative software development process. The study covered analysis of the entire development process of a specific program, carried out step by step in the AI-native IDE. Both the final result of the application and the progression of the successive stages of work were evaluated.

To enhance the reliability of the analysis, it was divided into three levels:

- the implementation level;
- the iteration level;
- the application level.

The implementation step level referred to a single feature or task, such as data validation. The iteration level covered the full cycle of implementing a given step, from the first iteration through to the acceptance of the feature. The entire application level referred to the execution of the project using a single model, from an empty repository to a working application with the specified scope of functionality implemented.

An iterative implementation process was adopted. The study designed a structured prompt system intended to ensure the comparability of results while preserving the nature of real-world work with AI models. With completely free interactions, differences in results could stem not only from the properties of a given model, but also from different ways of formulating commands, the order in which information is conveyed, and the scope of the user's clarifications.

The recording of results included the model's full response, the generated code, the number of iterations required to reach a correct solution, the task completion time, the types and number of errors encountered, the

occurrence of regression, and information on whether the step was passed or failed. One iteration was counted as a full cycle comprising a single prompt to the model, the model's response, the implementation of the proposed changes, and their testing. This means that both the initial task prompt and each correction prompt increased the iteration counter for a given step by one.

A limit of five iterations was set for each implementation step. If, after the fifth iteration, the code still did not meet the pass criteria, the step was considered failed in that experimental run. This limit was introduced to prevent the experiment from being extended uncontrollably when a model was unable to produce a working functionality and continued to generate ineffective corrections. A similar approach is used in studies on iterative repair of code generated by language models, where the number of repair attempts is limited to preserve the controlled character of the experiment [11]. In this study, the value of five iterations was adopted as a compromise between allowing the model to correct errors and maintaining the practical and comparable nature of the experiment.

The duration of a step was calculated from the moment the first task prompt was sent until the functionality was passed or the iteration limit was reached.

The complete experimental procedure is shown in Figure 2.

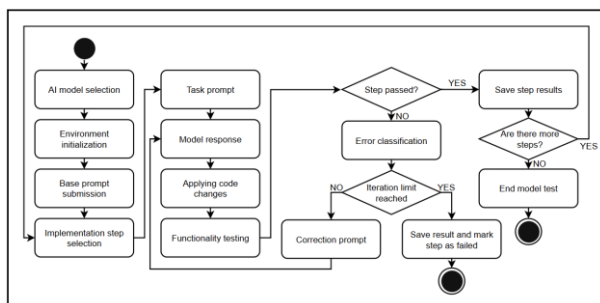


Figure 2: Experimental procedure for evaluating AI models

3.4. Prompt system

The system distinguishes between three types of prompts:

- base prompt;
- task prompt;
- correction prompt.

A base prompt was used at the beginning of work with each model and served as a common starting point. Its purpose was to convey:

- the general context of the project;
- the application's goal;
- a set of functional requirements;
- the adopted technology stack;
- the rules for conducting the experiment.

Task prompts were the main element of interaction with the models; each of these queries corresponded to a single functionality or a logically distinct part of the application. Task prompts included:

- the name of the current step;
- the expected behaviour;

- detailed requirement;
- technical constraints;
- a description of the required result.

Correction prompts were used only when the code returned by the model did not meet the criteria for passing a given step. Their role differed from that of task-based queries. They were not used to develop new functionalities, but to precisely eliminate errors encountered during testing. A prompt template was prepared that included:

- the name of the current step;
- a description of the expected behaviour;
- a description of the current behaviour;
- restrictions regarding the scope of changes;
- an error message, if any.

The prompt schema was designed to ensure the comparability of models on the one hand, while not unduly limiting the possibilities for proposing implementation solutions on the other.

The experiment consisted of 15 evaluation scenarios. The first scenario was the base prompt, which did not correspond to a separate application functionality but was included in the evaluation because it determined whether the model correctly understood the task context and followed the constraints of the experiment. The remaining 14 scenarios concerned the step-by-step implementation and refactoring of the Open Data Explorer application. The scenarios and their acceptance criteria are presented in Table 3.

Table 3: Evaluation scenarios and acceptance criteria

No.	Evaluation scenario	Acceptance criterion
1	Base prompt and project context	The model correctly follows the project context, constraints, and technology stack.
2	Project initialization	A runnable Streamlit application skeleton is created.
3	File loading	CSV, XLS, and XLSX files are correctly loaded into a pandas Data-Frame.
4	Input validation	Invalid or empty files are detected without application failure.
5	Data preview	The first rows of a valid dataset are displayed.
6	Dataset structure	Rows, columns, and data types are shown.
7	Descriptive statistics	Numerical statistics are calculated and displayed.
8	Missing values analysis	Missing values are counted and presented as numbers and percentages.
9	Column selection	Proper column selectors for visualisations are available.
10	Histogram	A labelled histogram is generated for a numerical column.
11	Bar chart	A labelled bar chart is generated for a categorical column.
12	Scatter plot	A labelled scatter plot is generated for two numerical columns.
13	Basic PDF report	A downloadable PDF report with data summary and analysis is generated.
14	Charts in PDF report	Available charts are correctly included in the PDF report.
15	Refactoring and modularity	The code is divided into logical modules without changing application behaviour.

3.5. Model evaluation metrics

The evaluation of artificial intelligence models was conducted using a set of metrics that assessed both the final outcome and the implementation process. The use of several groups of metrics was necessary because a model supporting programming can produce a correct result in various ways: quickly and stably, after many revisions, with readable code, or with a solution that is difficult to further develop.

The set of metrics was selected to enable the verification of research hypotheses. The study identified five groups of metrics presented in Table 4.

Table 4: Evaluation metrics used in the experiment

Metric	Description
Functional effectiveness	Indicated how well the model was able to implement the functionality in accordance with the requirements specified in that step during its first iteration.
Process efficiency	Determined the amount of work required to arrive at the correct solution.
Code quality	Assessed whether the generated solution was readable, modular, maintainable, and suitable for further development.
Project stability	Assessed the model's ability to develop the application without disrupting previously working functionalities.
Error analysis	Identified the types of problems that occurred most frequently when working with each model.

Functional effectiveness was assessed for each evaluation scenario on a scale of 1 to 5, where 1 indicated that the functionality had not been implemented, and 5 indicated its full implementation in accordance with the requirements. The assessment determined whether the model effectively met the requirements of each task prompt.

The efficiency of the process was assessed based on the number of iterations required to complete a given step and the time taken to complete that step. Two models might achieve the same result, but one might require significantly more revisions.

Code quality was evaluated by three independent experts based on code readability, solution modularity, simplicity of implementation, adherence to Python best practices, error handling, and the absence of redundant or dead code. The evaluation used a scale from 1 to 5, where 1 indicated very low or zero quality in a given area, and 5 indicated very good quality and the absence of noticeable or significant issues.

Project stability was assessed based on the number of functional regressions and the need to rework previously approved components. A regression was defined as a situation in which adding a new piece of code caused another feature to stop working.

The error analysis was conducted to identify the types of problems that occurred most frequently when working with specific models. The study identified the following types of errors:

- syntax errors;
- runtime errors;
- logical errors;

- integration errors;
- functional errors.

A syntax error indicated an incorrect code structure that prevented it from running. A runtime error indicated an exception that occurred during the application's operation. A logical error occurred when the application ran correctly but returned an incorrect result or performed an operation that did not follow the expected logic. An integration error indicated a conflict between newly generated code and previously functioning parts of the application. A functional error indicated that some of the requirements specified in a given implementation step were not met.

4. Experimental Results

4.1. Functional effectiveness of the models

The functional performance results are summarized in Table 5 and Figure 3. Claude achieved the highest score, scoring 70 out of a possible 75 points during its first iteration. ChatGPT scored 65 points, while SWE scored 53 points. The difference between the highest-scoring and lowest-scoring models was 17 points.

Table 5: Functional effectiveness scores of the compared models

Model	Total points
ChatGPT	65
Claude	70
SWE	53

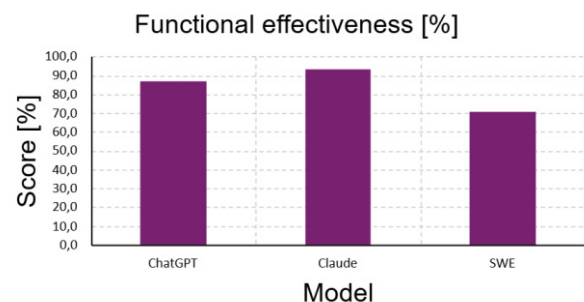


Figure 3: Functional effectiveness of the compared AI models

4.2. Process efficiency

The results regarding the number of iterations and the experiment completion time are presented in Table 6. Claude required the fewest iterations, completing the experiment after 18 iterations. ChatGPT required 22 iterations, and SWE required 26 iterations. Claude also achieved the shortest total execution time, which was 73:46 min. The times achieved by ChatGPT and SWE were similar, at 83:20 min and 83:14 min, respectively. Figure 4 shows the execution time of each model in individual implementation steps during the experimental run.

Table 6: Process efficiency of the compared models

Model	Number of iterations	Duration of the experiment (min:s)
ChatGPT	22	83:20

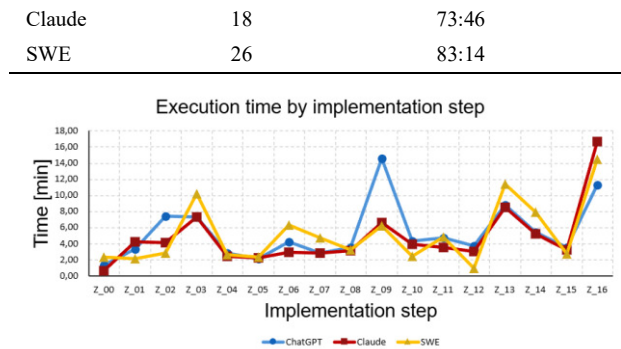


Figure 4: Execution time by evaluation scenario for each model

4.3. Code quality

The average code quality scores before and after refactoring are shown in Figure 5. In all cases, the score after refactoring was higher than before. Claude achieved the highest scores, with its average score increasing from 3.61 to 4.33. ChatGPT saw an increase from 3.22 to 4.00, while SWE increased from 2.17 to 3.34.

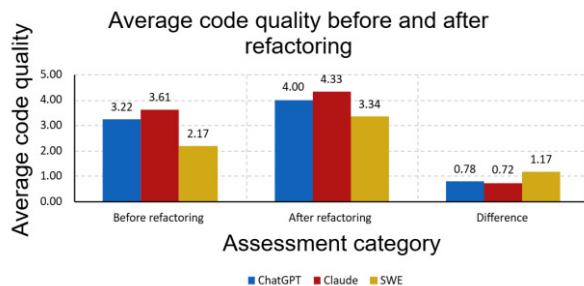


Figure 5: Average code quality before and after refactoring

4.4. Stability

During the experiment, two cases of functional regression were observed. One instance occurred in the ChatGPT model, and the other in the SWE model. No regression was observed in the Claude model. The results indicate that all models maintained sufficient stability to complete the experiment.

4.5. Error Analysis

The number of errors classified according to the established categories is presented in Table 7. The highest number of errors was recorded for the SWE model, which generated a total of 13 errors. ChatGPT generated 9 errors, and Claude generated 4 errors. No syntactic or logical errors occurred in any of the runs. The most common category was functional errors.

The percentage of errors attributable to each model is shown in Figure 6.

Table 7: Error types observed during the experiment

Error type	ChatGPT	Claude	SWE
Syntax	0	0	0
Runtime	1	0	3
Logic	0	0	0
Integration	2	1	1
Functional	6	3	9

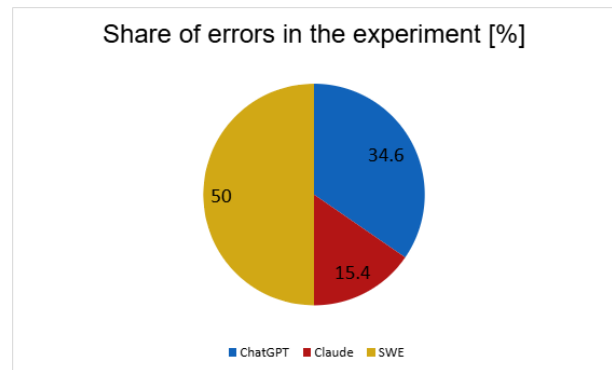


Figure 6: Share of errors generated by each model during the experiment

5. Results discussion

The results of the study confirmed the validity of hypothesis H1. The analysed models differed in terms of functional effectiveness, i.e., the degree to which the required application functionalities were correctly implemented. These differences indicate that the Claude model most frequently met the functional requirements in accordance with the specifications provided in the prompt, earning the maximum number of points per evaluation scenario. This model lost only 5 points relative to the maximum, meaning that in most stages it generated a complete solution or one requiring only minor corrections. Despite an equally high score, ChatGPT lost points on tasks requiring greater precision in implementation. The SWE model had the most significant issues with functionality and lost as many as 22 points. The model returned code that did not meet all functional requirements in 9 out of 15 experimental steps.

The difference in application development efficiency supports the findings of Hypothesis H2. Once again, Claude proved to be the most efficient, as it required the fewest correction requests and achieved the shortest total development time. The ChatGPT model required fewer iterations than SWE to generate the application, but the execution time for both models was very similar, which may have been due to the fact that the OpenAI model was tested first. Its execution required refining the method of testing and verifying the application, which may have reduced the time required for these steps when testing subsequent models. This also clearly demonstrates that the number of iterations does not fully reflect the time required for the entire process.

Hypothesis H3 is supported by the fact that higher functional effectiveness did not always translate to the highest code quality. The application could function correctly even though its structure was not optimal in terms of modularity and maintainability. Refactoring the code improved quality in all cases studied. Claude achieved the best final structure, while the ChatGPT and SWE projects retained greater limitations.

Hypothesis H4 was partially confirmed. Regressions occurred rarely; only two cases were recorded, and they did not significantly affect the application's performance. In the case of ChatGPT, the regression occurred

during the visualization phase, and in SWE, during the refactoring process. This means that problems occurred primarily in steps where the application was being expanded with more complex features or through a complete overhaul of the application. Despite the regressions, all models demonstrated high stability.

Hypothesis H5 was also confirmed. The models differed in the number and types of errors they produced. Functional errors were the most common, usually resulting from incomplete fulfilment of the requirements for a given step. They frequently appeared in models during similar steps of the experiment. This may indicate that some queries were more prone to misinterpretation or required rephrasing. Steps related to data validation, visualisations, and PDF generation proved particularly problematic. Integration errors occurred rarely and did not significantly affect the application's functionality, but they were noted in every model. Runtime errors occurred in ChatGPT and SWE and blocked the application's functionality. No syntax or logical errors occurred in the study. The absence of syntactic errors indicates that all models generated code that was correct in terms of the basic structure of the Python language. The absence of logical errors, on the other hand, means that no situations were observed in which the application functioned correctly but returned incorrect data analysis results.

6. Conclusions

The study evaluated selected artificial intelligence models in terms of their ability to generate a functional application and the course of the entire implementation process. The analysis covered functional effectiveness, work efficiency, code quality, project stability, and the number and types of errors. The scope of the work did not include training, fine-tuning, or internal analysis of the architecture of language models. The study focuses on their practical application in a programming environment during the development of a specific web application.

The results of the experiment show that the choice of AI model had a significant impact on the application development process. The models differed not only in terms of the implementation status of the required functionalities but also in the number of necessary corrections, development time, and the quality and stability of the solution. The choice of model can determine whether it will hinder or facilitate the programmer's work. The Claude model achieved the best results across all metrics. ChatGPT achieved intermediate results in most cases, while SWE achieved the lowest.

The results obtained should not be treated as a universal ranking of artificial intelligence models. The study was conducted on a single project, in a specific environment, and using a specific set of queries. For this reason, the results should be interpreted within the context of the research project. If even a single element were changed for example, a different test application—the results could be completely different.

In future research, it would be worthwhile to expand this experiment to include a larger number of models, more complex projects, and other programming technologies. It would also be necessary to incorporate automated unit tests into the study and expand the metrics analysed to include, for example, security assessments.

References

- [1] S. Bistarelli, M. Fiore, I. Mercanti, M. Mongiello, Usage of Large Language Model for Code Generation Tasks: A Review, *SN Computer Science* 6 (2025) 673, <https://doi.org/10.1007/s42979-025-04241-5>.
- [2] Y. Li, Y. Peng, Y. Huo, M. R. Lyu, Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context, In 2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code) IEEE (2024), <https://doi.org/10.1145/3643795.3648392>.
- [3] A. Kaplunovich, Transforming Code Development with Large Language Models, In 2025 International Conference on Intelligent Computing, Communication, Networking and Services (ICCNS) IEEE (2025), <https://doi.org/10.1109/ICCNS66249.2025.11428578>.
- [4] E. Mavridou, E. Vrochidou, T. Kalampokas, V. Kanakaris, G. A. Papakostas, AI-Powered Software Development: A Systematic Review of Recommender Systems for Programmers, *Computers* 14(4) (2025) 119, <https://doi.org/10.3390/computers14040119>.
- [5] K. T. Le, A. Andrzejak, Rethinking AI Code Generation: A One-Shot Correction Approach Based on User Feedback, *Automated Software Engineering* 31 (2024) 60, <https://doi.org/10.1007/s10515-024-00451-y>.
- [6] A. Sergeyuk, I. Zakharov, E. Koshchenko, M. Izadi, Human-AI Experience in Integrated Development Environments: A Systematic Literature Review, *Empirical Software Engineering* 31 (2026) 55, <https://doi.org/10.1007/s10664-025-10793-0>.
- [7] Z. Liu, Y. Tang, X. Luo, Y. Zhou, L. F. Zhang, No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT, *IEEE Transactions on Software Engineering* 50(6) (2024) 1548–1584, <https://doi.org/10.1109/TSE.2024.3392499>.
- [8] S. Li, Y. Cheng, J. Chen, J. Xuan, S. He, W. Shang, Performance Analysis of AI-Generated Code: A Case Study of Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder Models, *Empirical Software Engineering* 31 (2026) 62, <https://doi.org/10.1007/s10664-025-10776-1>.
- [9] W. Ito, Y. Obara, M. Sato, K. Kuramitsu, Hallucination Detection on Code Generation with SelfCheckGPT, *Journal of Information Processing* 33 (2025) 487–498, <https://doi.org/10.2197/ipsjip.33.487>.
- [10] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, G. Antoniol, Bugs in Large Language Models Generated Code: An Empirical Study, *Empirical Software Engineering* 30 (2025) 65, <https://doi.org/10.1007/s10664-025-10614-4>.
- [11] G. Dolcetti, V. Arceri, E. Iotti, S. Maffei, A. Cortesi, E. Zaffanella, Helping LLMs Improve Code Generation Using Feedback from Testing and Static Analysis,

- Discover Artificial Intelligence 6 (2026) 314, <https://doi.org/10.1007/s44163-026-01009-5>.
- [12] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. M. Jiang, GitHub Copilot AI Pair Programmer: Asset or Liability?, Journal of Systems and Software 203 (2023) 111734, <https://doi.org/10.1016/j.jss.2023.111734>.
- [13] L. Banh, F. Holldack, G. Strobel, Copiloting the Future: How Generative AI Transforms Software Engineering, Information and Software Technology 183 (2025) 107751, <https://doi.org/10.1016/j.infsof.2025.107751>.
- [14] H. Hao, K. A. Hasan, H. Qin, M. Macedo, Y. Tian, S. H. H. Ding, A. E. Hassan, An Empirical Study on Developers' Shared Conversations with ChatGPT in GitHub Pull Requests and Issues, Empirical Software Engineering 29 (2024) 150, <https://doi.org/10.1007/s10664-024-10540-x>.
- [15] Windsurf, Windsurf Editor, <https://windsurf.com/editor>, [11.05.2026].