Beyond Generation: A Comparative Analysis of Quality Issues in ChatGPT-Produced Code Across Programming Languages

Alexander J. Habegger
Miami University
Oxford, Ohio, USA

ABSTRACT

The widespread adoption of generative AI in software development has led to a need for code quality analysis. This paper introduces a novel approach to evaluating the code quality issues innate in code generated by ChatGPT, utilizing a comprehensive set of linters across multiple programming languages. Focusing on Python, Java, and JavaScript, our study systematically identifies and categorizes expected quality issues ranging from stylistic inconsistencies to more severe logical and fatal errors. Utilizing data extracted from the DevGPT database, we employ a variety of linters, including Pylint for Python, PMD for Java, and JSHint for JavaScript, to analyze the generated code. Our methodology also highlights each linter's strengths and weaknesses and reveals underlying patterns in the quality issues detected. This comparative analysis offers insights into error with generated cod to improve code quality.

Additionally, our conclusions contribute to the wider understanding of the limitations and potentials of AI-generated code. By shedding light on the common pitfalls and providing data-driven recommendations for best practices, this project aims to enhance the reliability and efficiency of AI as a developer tool. This work is part of the broader project to improve code quality in the era of AI-assisted programming, supporting the goal of the MSR 2024 Mining Challenge.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **General and reference** → *Empirical studies*; • **Information systems** → Data mining.

KEYWORDS

Code Generation, ChatGPT, Code Quality Analysis, Static Analysis

ACM Reference Format:
Alexander J. Habegger. 2024. Beyond Generation: A Comparative Analysis of Quality Issues in ChatGPT-Produced Code Across Programming Languages. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In software development, the emergence of generative artificial intelligence (AI) technologies has triggered a significant paradigm shift. These technologies offer promising avenues for accelerating development processes by generating code and tackling complex programming problems [9]. With GPT-4 demonstrating their capacity to pass assessments in higher education programming courses [8]. However, merging AI-generated code into development workflows introduces new challenges concerning code quality. Ensuring AI-generated code reliability, efficiency, and maintainability is paramount. Assessing the code quality produced by models like ChatGPT is essential for leveraging their full potential in software engineering.

Our research is primarily motivated by the desire to harness the power of generative AI, like ChatGPT, in software development. While these technologies promise unprecedented efficiency in automating routine coding tasks, their integration raises pertinent questions about software reliability and maintainability [1]. The key to successful integration is ensuring high-quality code that adheres to coding standards is optimized and free from bugs or logical errors. Our study systematically analyzes common quality issues in ChatGPT-generated code and provides actionable insights for enhancing AI-assisted code quality.

Our study focuses on a systematic analysis of the common code quality issues found in ChatGPT-generated code. With the proliferation of AI in software development, the need for quality assurance mechanisms has become more apparent. Linters are one solution. They are tools designed to flag programming errors, bugs, and stylistic mistakes, and serve as the main tool of our analysis. We employ a set of industry-standard linters tailored to multiple programming languages: Python, Java, and JavaScript. Our research uses data from the DevGPT database, a repository of ChatGPT-generated code samples. Their widespread use and relevancy drives the choice of programming languages and linters. For Python, we utilize Pylint; for Java, we utilize PMD; for JavaScript, we utilize JSHint. This selection covers various code quality aspects, from coding standards and conventions to potential bugs and optimization opportunities. By analyzing the feedback from these linters, we aim to uncover patterns in the quality issues of AI-generated code, providing insights into the strengths and weaknesses of each tool in addressing these challenges. Ultimately, this study aims to bridge the gap between the potential of AI as a developmental aid and the practical realities of ensuring high-quality software production, aligning with the objectives of the MSR 2024 Mining Challenge.

2 RELATED WORKS

Integrating generative AI technologies into software development has been a subject of significant academic interest, with a particular focus on code generation and analysis. We will review recent studies that explore various dimensions of AI-assisted coding, focusing on generative models and their implications for code quality, efficiency, and correctness. The metrics are also used to test code quality. The works provide a foundation for our research: Poldrack et al. (2023) [7] conducted experiments with GPT-4 to explore AI-assisted coding. The study demonstrates the potential of large language models in automating coding tasks and enhancing developer productivity. The research offers insights into the capabilities of GPT-4 in providing benefits to developers when concerns of code quality are mitigated. Idrisov and Schlippe (2024) [4] provides an overview of the methodologies and tools employed in automated code generation. Their study assesses the effectiveness of generative models in creating functional code across various programming languages. The comparison of our results across languages and theirs will help contribute to the broader understanding of AI's role in software development and its effects on and understanding different programming languages. Huang et al. (2024) [3] introduced a tool called EffiBench for a benchmarking framework to evaluate the efficiency of automatically generated code. Efficiency is particularly relevant to our project as it plays an important role in code quality. The tool offers a systematic approach to measuring the performance of AI-generated code. EffiBench's findings underscore the importance of optimizing AI-generated code for practical applications. It also shows the importance of clearly defined metrics. Yetiştiren et al. (2023) [10] evaluated the code quality comparing AI-assisted code generation tools, including GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. Their comprehensive assessment of these tools' output provides valuable benchmarks for code quality, which inform our investigation into the quality issues present in ChatGPT-generated code. This study highlights the variability in code quality across different AI-assisted tools. Then, Liu et al. (2024) [6] evaluated large language models for code generation, questioning the correctness of code produced by ChatGPT. The precise methodology contributes to the resources at our disposal to test the reliability of AI-generated code, emphasizing the need for rigorous testing and validation to ensure code correctness.

Moreover, the role of AI in enhancing static analysis is explored by Li et al. (2023) [5], who investigates how large language models can assist in reducing inaccuracies in bug-finding tools. Their findings demonstrate the potential of AI in minimizing false positives and negatives, thereby contributing to more reliable static analysis. Complementing this, Guo et al. (2024) [2] explore the application of ChatGPT in automating code refinement tasks in code reviews. They compare ChatGPT's performance against traditional tools, illustrating its superior efficiency and accuracy in code refinement.

Together, these studies form a backdrop against which our research is positioned. The studies highlight the increasing dependence on generative AI for code generation and the challenges of ensuring AI-generated code quality, efficiency, and correctness. Our work is based on these insights, explicitly focusing on using linters to detect and mitigate quality issues.

3 METHODOLOGY

3.1 Datasets

In our study, we employ two distinct datasets: the DevGPT dataset and a dataset gathered using the GitHub API.

DevGPT Dataset. The DevGPT dataset consists of 1,831 AI-generated code snippets sourced from various platforms, including GitHub issues, pull requests, discussions, commits, and Hacker News threads. These snippets are collected across four daily snapshots and encompass code written in Python, Java, and JavaScript. 42,243 lines of code (LOC) from this dataset have been analyzed using industry-standard linters tailored to each programming language: Pylint for Python, PMD for Java, and JSHint for JavaScript.

GitHub API Dataset. The GitHub API Dataset comprises 278 files randomly selected from 278 GitHub repositories, characterized as "small projects," each with between 2 and 100 stars to limit the amount of ultra-refined code. Similar to the DevGPT dataset, this dataset includes files in Python, Java, and JavaScript, amounting to 25,351 LOC, which has been linted to assess code quality.

3.2 Research Questions

Two primary research questions guide our study:

- **RQ1: How does AI-Generated Code Perform Across Programming Languages?** We analyze the AI-generated code from the DevGPT dataset to evaluate the performance of generative AI in coding tasks across different programming languages. Using linters specific to each language, we categorize and analyze the prevalent issues in the code, focusing on syntactic errors, logical flaws, and inefficient practices.
- **RQ2: What is the Difference Between AI-Generated and Human-Written Code?** We compare the AI-generated code from the DevGPT dataset with the human-written code from GitHub to discern differences in code quality based on error prevalence and error type. This comparative analysis employs quantitative metrics derived from our linting results, facilitating an analysis of the disparities in code quality.

3.3 Data Analysis Plan

Our analysis involves a detailed examination of linting results from both datasets. We aggregate and categorize errors identified by the linters, focusing on their type and frequency. The error type is determined by a classification system that is articulated in Appendix A. The frequency is based on errors per LOC, which is the number of errors that the linter found divided by the number of lines of code excluding whitespace. Another metric that is used is the error-free rate. The error-free rate refers to the percentage of code snippets where the linters found no code errors.

We will acknowledge potential biases in our dataset and the limitations of linters, as they might not capture all nuances of code quality. We expect to uncover insights into the quality of AI-generated code and understand ChatGPT's limitations and potential in software development. This will contribute to the broader conversation on integrating AI tools in code generation while maintaining high

code quality standards. The code used will be publically available at this location: github.com/ahabegger/CodeQuality4DevGPT.

4 RESULTS

4.1 RQ1: Performance of AI-Generated Code Across Programming Languages

We evaluated the code quality of AI-generated code in Python, Java, and JavaScript based on metrics such as error rate per line of code (LOC) and the percentage of error-free code snippets. Our findings for each programming language are detailed below.

The dataset included 759 Python snippets, amounting to 17,331 LOC, averaging 22.8 LOC per snippet. We identified 2,394 errors, indicating an alarmingly high error rate of 138.1 per 1,000 LOC. This high error rate underscores the challenges in maintaining syntactic or semantic correctness, with only 1.8% of snippets being error-free.

Java snippets totaled 189, covering 4,736 LOC with an average of 25.1 LOC per snippet. These snippets showed a significantly lower error rate of 27.2 per 1,000 LOC, with 129 errors detected in total. Notably, 41.8% of Java snippets were error-free, reflecting a higher robustness and reliability in AI-generated Java code compared to Python, indicating potential areas for improvement in AI programming models for Python.

There were 883 JavaScript snippets, totaling 20,176 LOC, averaging 22.8 LOC per snippet. These snippets exhibited 1,077 errors, yielding an error rate of 53.4 per 1,000 LOC. However, 72.1% of the snippets were error-free, suggesting that AI-generated JavaScript aligns more closely with standard practices and avoids common errors more effectively than Python. This higher alignment and effective error avoidance in JavaScript highlight the strengths of AI in this language and suggest potential areas for improvement in other languages.

> Python code generated by AI exhibits a high error rate and a low percentage of error-free snippets, while Java and JavaScript show greater robustness. However, AI models seem particularly well-tuned to JavaScript's coding standards and nuances, suggesting their adaptability and potential for better alignment with language-specific features or community coding standards.

We analyzed the types of errors encountered by AI-generated code in Python, Java, and JavaScript to identify specific weaknesses in ChatGPT code for each language. See Table 1 and Figure 1 for the breakdown:

Table 1: Error Distribution in AI-Generated Code Across Programming Languages

Error Type	AI Python %	AI Java %	AI JavaScript %
Syntax and Formatting	22.64%	0.00%	40.30%
Variable and Name	25.15%	20.93%	18.20%
Import	12.41%	25.58%	1.11%
Control Flow and Logic	1.38%	14.73%	1.49%
Function and Method	3.97%	34.11%	4.46%
Documentation	34.09%	4.65%	4.27%
Other	0.38%	0.00%	30.18%

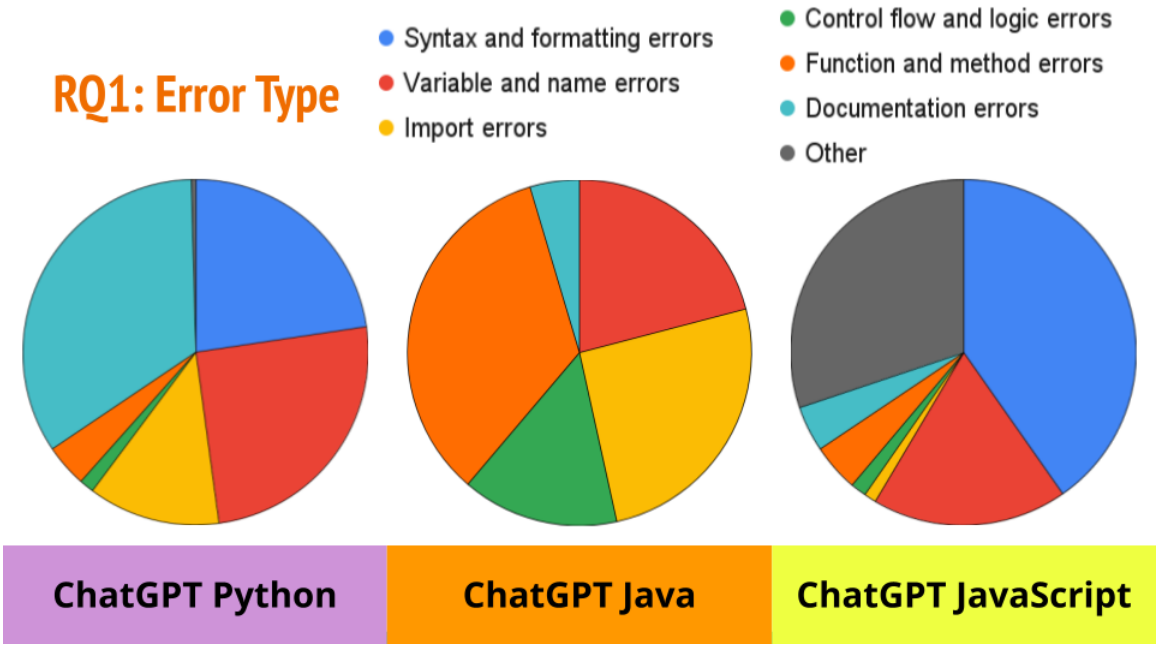

Figure 1: Distribution of error types across programming languages

This error analysis underscores distinct challenges faced by AI across different languages. Python's AI-generated code primarily struggles with documentation and naming conventions, while Java shows better syntax adherence but has difficulties with function/method definitions and imports. JavaScript's high rate of syntax and formatting errors suggests gaps in the AI's understanding of the language's flexible syntax. The significant other category in JavaScript could indicate difficulties in handling less conventional or more complex functionalities. These findings suggest promising ways to refine AI models tailored to each programming language's nuances.

> Python's AI-generated code frequently encounters documentation and variable naming errors. While adhering to syntax well, Java needs help with function/method definitions and import management. However, JavaScript needs an improved understanding of its flexible syntax rules.

4.2 RQ2: AI-Generated Code Compared to Human-Written Code

This section evaluates the differences in quality between AI-generated code from the DevGPT dataset and human-written code from the GitHub dataset, focusing on error rates, total errors, and the prevalence of error-free snippets across Python, Java, and JavaScript.

Table 2: Comparison of Code Quality Metrics Across AI-Generated and Human-Written Code

Category	Average LOC per Snippet	Average Errors per 1000 LOC	% of Error Free Snippets
AI Python	22.8	138.1	1.8%
AI Java	25.1	27.2	41.8%
AI Javascript	22.8	53.4	72.1%
Human Python	104.7	85.8	6.2%
Human Java	48.8	23.1	12.5%
Human Javascript	85.4	101.6	44.8%

The AI-generated Python code, comprising 759 snippets totaling 17,331 LOC, exhibited a high error rate of 138.1 per 1,000 LOC and only 1.8% error-free snippets, highlighting substantial quality issues. In contrast, human-written Python code included 129 snippets with 13,508 LOC, showing a reduced error rate of 85.8 per 1,000 LOC and a higher percentage of error-free snippets at 6.2%, indicating superior code quality.

AI-generated Java code consisted of 189 snippets and 4,736 LOC with a relatively low error rate of 27.2 per 1,000 LOC and 41.8% of snippets error-free, suggesting effective AI coding practices for Java. However, human-written Java, while smaller in sample size with 24 snippets and 1,171 LOC, demonstrated an almost similar error rate of 23.1 per 1,000 LOC but only 12.5% error-free snippets, indicating marginally better human performance in error minimization over smaller samples.

AI-generated JavaScript showed 883 snippets and 20,176 LOC, with an error rate of 53.4 per 1,000 LOC and an impressive 72.1% of snippets being error-free. In comparison, human-written JavaScript comprised 125 snippets and 10,672 LOC, presenting a higher error rate of 101.6 per 1,000 LOC and 44.8% error-free snippets, showing that AI can generate cleaner code in larger samples.

> AI's proficiency in Java and JavaScript suggests a potential reduction in error rates and an increase in error-free code generation. In contrast, Python performs better with human coding, indicating a nuanced understanding of dynamic language intricacies.

In Python, AI-generated code exhibited diverse errors, with significant documentation (34.09%) and variable naming issues (25.15%). Conversely, human-written code primarily faced syntax and formatting issues (50.56%), suggesting AI's relative strength in syntactical correctness but weaker performance in nuanced aspects like documentation.

Table 3: Detailed Error Types in AI and Human Python Code

Error Type	AI Python %	Human Python %
Syntax and Formatting	22.64%	50.56%
Variable and Name	25.15%	0.43%
Import	12.41%	3.28%
Control Flow and Logic	1.38%	0.00%
Function and Method	3.97%	0.00%
Documentation	34.09%	8.80%
Other	0.38%	36.93%

For Java, both AI and human code showed a high adherence to syntactic standards, with AI-generated code having a balanced error distribution, especially in function and method definitions.

Table 4: Error Types in AI and Human Java Code

Error Type	AI Java %	Human Java %
Syntax and Formatting	0.00%	0.00%
Variable and Name	20.93%	14.81%
Import	25.58%	33.33%
Control Flow and Logic	14.73%	7.41%
Function and Method	34.11%	44.44%
Documentation	4.65%	0.00%
Other	0.00%	0.00%

AI-generated JavaScript code faced significant syntax and formatting challenges, a common issue also observed in human-written code, suggesting an area for further improvement in both AI and human coding practices.

Table 5: Error Types in AI and Human Javascript Code

Error Type	AI Javascript %	Human Javascript %
Syntax and Formatting	40.30%	83.21%
Variable and Name	18.20%	0.46%
Import	1.11%	0.09%
Control Flow and Logic	1.49%	0.09%
Function and Method	4.46%	0.55%
Documentation	4.27%	0.92%
Other	30.18%	14.67%

This comparison not only underscores significant differences in error types between AI-generated and human-written code but also illustrates AI's ability to handle syntax and formatting effectively, particularly in Python and Java. However, the analysis reveals AI's deficiencies in more nuanced aspects like documentation and variable naming, where human code shows relative strength. These observations suggest potential areas for enhancing AI coding models to improve their understanding and generation of more complex and nuanced code constructs.

> AI models excel in producing syntactically correct code, particularly in Python and Java. However, they lag in nuanced areas like documentation and variable naming, highlighting the need for targeted improvements in AI's understanding of these aspects.

5 IMPLICATIONS

5.1 Suggestions for AI Developers

In their pursuit of enhancing code generation models, AI developers should emphasize their work's collaborative nature. By improving the training of these models, they can better handle the nuances of different programming languages, particularly those with high error rates like Python. Enhancing the model's capability to generate well-documented code and adhere to naming conventions could significantly improve the usability of generated code. Incorporating more comprehensive error detection and correction mechanisms can also reduce the frequency and severity of errors. Expanding training datasets to include a more comprehensive array of coding styles and scenarios could help AI models generalize better across real-world applications. This collaborative approach, involving the developer community to gather feedback and refine models iteratively, would further enhance the effectiveness and reliability of AI-generated code.

5.2 Suggestions for Developers Using AI

The role of software developers in integrating AI tools into coding practices is crucial. Their strategic approach is needed, especially in scenarios where the AI is known to produce higher error rates, such as Python. Thoroughly reviewing AI-generated code, ensuring it functions correctly and adheres to best practices for maintainability and scalability, is a key part of this process. Leveraging AI to generate boilerplate code and automate routine tasks can increase productivity, but human involvement in the coding process to ensure quality and correctness is essential. Providing feedback to AI tool developers can also help improve the tools, making them more useful in professional programming environments.

> For effective AI integration into software development, it is essential that AI developers continuously refine their models based on user feedback and evolving programming standards. Similarly, AI developers should adopt a balanced approach, leveraging AI for suitable tasks and rigorously validating.

6 THREATS TO VALIDITY

6.1 Internal Threats

- **Small Dataset:** Our analysis across different programming languages and the use of various linters might be affected by the relatively small size of the dataset. This limitation could influence our findings' statistical power and generalization, as larger datasets might exhibit different trends or reveal more nuanced insights.
- **Linter Classification System:** The classification system used by different linters may not be uniform, leading to inconsistencies in error categorization. Since linters are designed with specific frameworks and coding standards in mind, their outputs might vary significantly, potentially skewing the error analysis and the interpretation of AI-generated code quality.
- **Contextual Limitations of Generated Code:** ChatGPT and similar AI models often generate code snippets without complete contextual information, such as missing import statements. These omissions are classified as errors in our analysis but may be intentional to focus on the specific coding task requested by users. This discrepancy highlights a limitation in how AI understands and handles task boundaries and could affect the perceived error rate in AI-generated code.
- **Linting Failure Rate:** The failure rate of linting tools. Bugs or limitations in handling certain code constructs (usually UTF-8 characters) threaten our analysis's validity. If linters fail to parse or assess the code correctly, this could lead to incorrect error reports, affecting our conclusions regarding the quality of AI-generated code. Linting failures were thrown out of the study.

6.2 External Threats

- **Generalization Across AI Products:** Our study primarily focuses on code generated by ChatGPT. Consequently, assuming that our findings apply to all AI code generation products might not be valid. Different AI models may have distinct training datasets, algorithms, and optimization techniques, which can significantly impact the quality of the generated code.
- **Assumptions Across Programming Languages:** The study assumes a level of comparability across programming languages that may not exist. Each programming language has unique features, use cases, and community standards that can influence how AI models perform in generating code. The generalization of our findings across all languages without considering these specificities could misrepresent the capabilities and limitations of AI in code generation.

By acknowledging these internal and external threats, we can better frame the conclusions of our study within its appropriate context and limitations. Future research should address these threats by expanding the dataset, refining the analysis techniques, and broadening the scope to include multiple AI code generation models and a more comprehensive range of programming languages.

7 CONCLUSION

This study has provided a comprehensive analysis of the quality issues prevalent in AI-generated code, with a particular focus on code produced by ChatGPT in Python, Java, and JavaScript. Our findings have highlighted AI's diverse capabilities and limitations in code generation, revealing distinct patterns of strengths and weaknesses across different programming languages. Some of the **Key Findings** were:

- Python Code Generation: AI-generated Python code exhibited high error rates, particularly in the documentation and naming conventions, suggesting challenges in handling dynamic programming language nuances.
- Java Code Generation: AI performed relatively better in generating Java code, with lower error rates and a higher percentage of error-free snippets, indicating more vital adherence to syntactic norms.
- JavaScript Code Generation: Despite some challenges with syntax and formatting, JavaScript code generated by AI was primarily compliant with standard practices, resulting in a significant proportion of error-free snippets.

The discrepancies in error types and rates among the programming languages underscore the need for targeted improvements in AI models, especially for dynamically typed languages like Python. The study also revealed that while AI can efficiently handle certain aspects of code generation, such as syntax and basic structure, it needs help with more complex elements like contextual understanding and advanced logic integration.

To build on the findings of this study, future research should focus on several key areas. Larger and more diverse datasets should be created to enhance the robustness and generalizability of the results. This expansion would help understand AI's performance variations under different coding scenarios and requirements. Further studies could develop more sophisticated error analysis techniques that consider the contextual appropriateness of AI-generated code, potentially adjusting linter tools to better evaluate the functional correctness of code rather than just syntactic conformity. Comparing different AI code generation models could provide deeper insights into the specific training and architecture choices, contributing to better code quality across various languages. Investigating the integration of AI tools with human oversight could offer valuable data on combining human expertise with automated code generation to optimize productivity and code quality.

In conclusion, while AI demonstrates substantial potential to assist in software development, its effective utilization requires careful consideration of its limitations and capabilities. By addressing the identified issues and continuously refining AI models based on developer feedback and advanced testing, AI can become an even more powerful tool in the programming arsenal.

REFERENCES

[1] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[2] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[3] Dong Huang, Jie M Zhang, Yuhao Qing, and Heming Cui. 2024. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. *arXiv preprint arXiv:2402.02037* (2024).

[4] Baskhad Idrisov and Tim Schlippe. 2024. Program Code Generation with Generative AIs. *Algorithms* 17, 2 (2024), 62.

[5] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2107–2111.

[6] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[7] Russell A Poldrack, Thomas Lu, and Gašper Beguš. 2023. AI-assisted coding: Experiments with GPT-4. *arXiv preprint arXiv:2304.13187* (2023).

[8] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. *arXiv preprint arXiv:2306.10073* (2023).

[9] Tom Taulli. 2023. Auto Code Generation: How Generative AI Will Revolutionize Development. In *Generative AI: How ChatGPT and Other AI Tools Will Revolutionize Business*. Springer, 127–143.

[10] Burak Yetiştiren, Işık ¨Ozsoy, Miray Ayerdem, and Eray T¨uz¨un. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023).

APPENDIX A

Classification System for Code Analysis

The classification system for analyzing the code to understand the specific types and frequencies of errors across different programming languages. This appendix provides a detailed look at how errors are categorized in Java, JavaScript, and Python. The No Package error in Java was discarded due to the contextual limitations of the linters.

Table 6: Python Error Classification

Syntax and Formatting Errors	Variable and Name Errors	Import Errors	Control Flow and Logic Errors	Function and Method Errors	Documentation Errors	Other Errors
Trailing whitespace	Undefined variable	Wildcard import	Return outside function	Too many locals	Missing docstring	Unaccounted Errors
Bad continuation	Redefined builtin	Unused import	Used before assignment	Too few public methods		Other Errors
Syntax error	Invalid name	Import error	Pointless statement	Unused argument		
Parse error	Blacklisted name	Relative beyond top level	Bare except	No self use		
Line too long			Redefined outer name			
Anomalous backslash in string				Too many arguments		
Trailing comma tuple						
Trailing newlines						
Bad whitespace						

Table 7: Java Error Classification

Syntax and Formatting Errors	Variable and Name Errors	Import Errors	Control Flow and Logic Errors	Function and Method Errors	Documentation Errors	Other Errors
UnusedLocalVariable		UnnecessaryImport	UnnecessaryLocalBeforeReturn	UseLocaleWithCaseConversions	UncommentedEmptyConstructor	Other Errors
UnusedPrivateField			LiteralsFirstInComparisons	UnusedPrivateMethod		Uncategorized Errors
			EmptyControlStatement	UnusedFormalParameter		
			LooseCoupling			
				UseUtilityClass		

Table 8: JavaScript Error Classification

Syntax and Formatting Errors	Variable and Name Errors	Import Errors	Control Flow and Logic Errors	Function and Method Errors	Documentation Errors	Other Errors
Missing semicolon	Missing initializer for constant	General Import Error	Operator Error	Unlocal Variable Call in Function	Dot Notation	Other Errors
Unexpected comma	Redeclaring Variables	Module Code Error		Class properties Outside Methods		Uncategorized Errors
Unnecessary semicolon	Overiding Constants			Duplicate Class Methods		
Unclosed regular expression	Used before Declared					
String Formatting	Use Restricted Word					
Confusing Syntax	Incorrect Identifier					
Closing Pair not Used						
Labelling Error						