Evaluating Code Quality Generated in Large Language Models: A Multi-Language Empirical Study

تقييم جودة الشفرة البرمجية المولدة في النماذج اللغوية الكبيرة
(دراسة تجريبية متعددة اللغات)

Dr. Rasha Abdulaziz Bin-Thalab[1], Osamah Abdullah Abduljalil[2]
[1]Associate Professor, Department of Computer Engineering, College of Engineering and Petroleum, Hadhramout University, Yemen
[2]Master's student, Computer Science Department, College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University, Saudi Arabia
[1]r.binthalab@hu.edu.ye, [2]osamah.abduljalil@gmail.com

Received: 10/10/2025 Accepted: 5/11/2025

ABSTRACT

Software engineering has undergone a significant transformation due to the rapid development of Large Language Models (LLMs). Modern LLMs like GitHub Copilot, GPT-3.5, and GPT-4 are becoming more adept in generating useable source code in a range of programming languages with little human intervention. However, there is still debate over their outputs' internal quality, maintainability, and documentation. The study uses three popular programming languages—Python, Java, and JavaScript—to empirically evaluate how successfully LLMs generate code for issues of varying difficulty. SonarQube was used to examine the generated code and quantify cognitive difficulty, cyclomatic complexity, code smells, and comment ratios after the collection of a dataset of algorithmic tasks and independently solving the data via each model. The findings demonstrate that GPT-4 regularly creates code that is easier to maintain and understand than previous models, while Copilot consistently produces more comments but less structurally sound code. GPT-3.5 performs mediocrely and has moderate variability.

Key Words:
- **Large Language Models**
- **Code Generation**
- **Software Quality**
- **Copilot**
- **GPT**

الملخص:

شهدت هندسة البرمجيات تحولًا كبيرًا بسبب التطور السريع لنماذج اللغة الكبيرة (LLMs). أصبحت نماذج اللغة الكبيرة الحديثة مثل GitHub Copilot و GPT-3.5 و GPT-4 أكثر مهارة في إنشاء شفرة مصدر قابل للاستخدام في مجموعة من لغات البرمجة مع تدخل بشري ضئيل. ومع ذلك، لا يزال هناك جدل حول الجودة الداخلية لمخرجاتها وقابليتها للصيانة والتوثيق. تستخدم هذه الدراسة ثلاث لغات برمجة شائعة — Python و Java و JavaScript — لتقييم مدى نجاح نماذج اللغة الكبيرة (LLMs) في إنشاء شفرات برمجية لمشكلات متفاوتة الصعوبة. تم استخدام SonarQube لفحص الشفرات التي تم إنشاؤها وقياس الصعوبة المعرفية والتعقيد الدوري

الكلمات المفتاحية:
- نماذج اللغة الكبيرة
- توليد الشفرة البرمجية
- جودة البرمجيات
- Copilot
- GPT

ومشاكل الشفرات ونسب التعليقات بعد أن تم جمع مجموعة بيانات من المهام الخوارزمية وحلها بشكل مستقل بواسطة كل نموذج. تظهر النتائج أن **GPT–4** ينتج بانتظام كودًا أسهل في الصيانة والفهم من النماذج السابقة، في حين ينتج **Copilot** باستمرار تعليقات أكثر ولكن كودًا أقل صلابة من الناحية الهيكلية. يقدم **GPT–3.5** أداءً متوسطًا ويتميز بتقلب معتدل.	

1. Introduction

In recent years, *Large Language Models (LLMs)* have evolved from general-purpose natural language generators into powerful tools capable of producing executable source code (Chang et al., 2024). These models—such as OpenAI's GPT-3 and GPT-4, GitHub Copilot, and CodeLlama—have demonstrated remarkable capabilities in transforming natural language prompts into syntactically correct and functionally valid codes. This technological leap has revolutionized software development practices by enabling *AI-assisted programming*, where developers can automate repetitive tasks, accelerate prototyping, and receive real-time coding suggestions. Consequently, LLMs are increasingly being integrated into integrated development environments (IDEs), educational tools, and professional workflows.

The promise of LLMs in software engineering stems from their large-scale training on code and natural language corpora, sophisticated Transformer architectures, and advanced fine-tuning techniques such as *instruction-tuning* and *in-context learning* (Ouyang et al., 2022). These capabilities allow them to generalize across programming paradigms, languages, and problem-solving contexts, offering significant potential for productivity gains and democratization of software development.

However, despite their growing adoption, the quality of LLM-generated code remains an open research question. Most prior evaluations of LLMs' coding abilities have focused primarily on *functional correctness*—whether the code executes and produces the expected outputs (Finnie-Ansley et al., 2023; Zhang et al., 2024). Yet, software quality encompasses far more than correctness. *Readability*, *understandability*, *maintainability*, and *code smells* are critical factors that directly impact long-term sustainability, debugging efficiency, and collaborative development (Dantas & Maia, 2021). Code that merely runs correctly but is complex, redundant, or poorly structured can hinder maintainability and reliability in production environments.

Existing empirical studies have also been limited in scope—often restricted to a single programming language (commonly Python or Java) or a single model, with evaluation metrics that do not fully capture multidimensional aspects of code quality ((Adamson & Bägerfeldt, 2023). Few works have explored how *problem difficulty* and *programming language diversity* influence the resulting quality of LLM-generated code. Moreover, comparative, cross-language analyses remain underexplored, leaving a gap in understanding how these models behave under varying complexity levels and coding paradigms.

To address these gaps, this study presents a systematic, empirical assessment of code generated by three prominent LLM-based tools—Copilot ((Chen et al., 2021), GPT-3

(Ouyang et al., 2022), and GPT-4 (Achiam et al., 2023) —across multiple programming languages (Java, Python, and JavaScript) and problem difficulty levels (low, medium, and high). A dataset of 99 programming problems was compiled and solved by these models. The resulting code samples were analyzed using *SonarQube*, a widely recognized static analysis tool, to evaluate quality indicators including *understandability*, *code smells*, *cognitive complexity*, and *maintainability index*. Through this analysis, the study provides deeper insight into how LLMs perform across different contexts and how the generated code aligns with professional software engineering standards.

This study makes the following key contributions:

1. **Comprehensive Multi-Dimensional Evaluation:**
   It presents a systematic empirical analysis of LLM-generated code quality across three major programming languages (Java, Python, and JavaScript) and three problem difficulty levels, filling a notable gap in cross-language and complexity-based evaluation.
2. **Integration of Automated Quality Assessment Metrics:**
   The study employs *SonarQube* to quantitatively assess multiple software quality indicators—such as understandability, code smells, cognitive complexity, and maintainability—providing an objective framework for evaluating LLM-generated code beyond mere functional correctness.
3. **Comparative Analysis of Prominent LLMs:**
   By comparing GitHub, Copilot, GPT-3, and GPT-4, the research identifies distinct behavioral and quality patterns among current LLM tools, revealing their relative strengths and weaknesses in generating sustainable code.
4. **Empirical Insights into Code Understandability:**
   The study provides one of the first large-scale analyses of *readability and understandability* of LLM-generated code, contributing to ongoing discussions on software quality and maintainability in AI-assisted development.

Research Questions (RQ)

The study focuses on the main research question:

RQ 1: How exemplary is the code quality of solutions generated by LLMs?

This leads to the following specific qualitative questions:

1. **RQ1.1 (Understandability):** How understandable is the code generated by LLMs, as measured by Cognitive Complexity and Cyclomatic Complexity?
2. **RQ1.2 (Maintainability):** How maintainable is the code generated by LLMs, as measured by the detection of "Code Smells"?
3. **RQ1.3 (Documentation):** What is the quality of the documentation in the code generated by LLMs, as measured by the Comment Ratio?

Using a number of analytical methods, this study aims to objectively evaluate the readability and maintainability of code generated by GPT-3.5, GPT-4, and Copilot. It specifically seeks to measure code understandability using Cognitive and Cyclomatic Complexity metrics and evaluate maintainability by identifying and categorizing common "code smells." The study also assesses the quality of documentation by examining the Comment Ratio and its correlation with problem difficulty. Finally, it conducts a statistical

assessment of these models' performance in producing understandable and maintainable code for various problem complexity levels and programming languages, particularly Python, Java, and JavaScript.

This research study is important for developers because it helps them understand the quality of code that LLMs should produce, which aids in code review and integration processes. Model developers, on the other side, will be able to improve future training efforts for these models by identifying structural flaws (like name and organization) that need to be fixed.

The remainder of this paper is organized as follows. Section 2 reviews the related literature on LLM-based code generation and software quality assessment. Section 3 describes the experimental setup, including dataset construction, model selection, and evaluation methodology. Section 4 presents the findings: results and analysis of the SonarQube-based evaluation. Section 5 discusses the implications of the findings for developers and model designers. Finally, Section 6 concludes the paper and outlines potential directions for future research.

2. Related Literature

Recent research has extensively explored the capabilities and limitations of Large Language Models (LLMs) in generating software code, particularly focusing on code quality dimensions such as readability, maintainability, reliability, and correctness. Dantas & Maia (2021) conducted an empirical study using senior Java developers to examine the relationship between automated code quality metrics and human comprehension. Their findings revealed that readability scores are significantly stronger predictors of human understanding than understandability metrics alone, highlighting the importance of readability in assessing perceived code quality.

Moradi Dakhel et al. (2023) evaluated GitHub Copilot as an AI pair programmer and found that while it can enhance productivity for expert developers, it often produces buggy or incomplete code that novice programmers may fail to detect. These contextual limitations lead to degraded code quality and potential propagation of subtle defects. Similarly, Yetiştiren et al. (2023) compared GitHub Copilot, Amazon CodeWhisperer, and ChatGPT using the HumanEval benchmark, integrating SonarQube metrics to assess maintainability, reliability, and security. ChatGPT achieved the highest correctness rate but also exhibited moderate levels of technical debt and code smells, underscoring trade-offs between accuracy and maintainability.

Ouh et al. (2023) analyzed ChatGPT’s performance in generating Java programming solutions, concluding that the model performs well on straightforward text-based problems but struggles with tasks involving non-textual artifacts or implicit contextual knowledge. Tosi (2024) further compared GPT-3.5, GPT-4, and Google Bard on complex Java problems, finding that although GPT-4 generated syntactically simpler and more correct code, human intervention was still necessary to ensure functional completeness, indicating the models’ limitations for autonomous code generation.

From a performance perspective, Coignion et al. (2024) demonstrated that LLM-generated solutions for LeetCode problems achieved runtime efficiency comparable to human-written code, though they cautioned about potential dataset contamination and benchmark instability. In a large-scale empirical study, Liu et al. (2024) evaluated over 4,000 Java and Python snippets produced by ChatGPT, reporting that although more than two-thirds were functionally correct, many suffered from poor maintainability and style issues, suggesting that correctness does not necessarily imply quality.

Sajadi et al. (2025) examined the security awareness of GPT-4, Claude 3, and Llama 3 finding that these models rarely issue proactive warnings about insecure code, exposing risks of integrating vulnerable LLM-generated snippets. Similarly, Jamil et al. (2025) compared human-written and LLM-generated Python code, revealing that LLMs often produce more reliable code for simple tasks but exhibit structural deficiencies and maintainability concerns in complex scenarios. Santa Molison et al. (2025) reinforced these observations, emphasizing that while fine-tuning reduces high-severity bugs, deeper design flaws persist in advanced tasks.

Finally, Simoes and Venson (2025) investigated the sensitivity of LLMs to deliberate readability modifications—such as renaming identifiers or removing comments—and found that LLMs' quality judgments degrade with reduced readability. This demonstrates that LLMs implicitly capture human-centric readability cues but remain inconsistent across runs. Collectively, these studies indicate that while LLMs demonstrate strong potential in generating functionally correct and efficient code, challenges persist regarding readability, maintainability, and contextual reliability, warranting further systematic investigation into their internal code quality characteristics.

Collectively, these studies underscore that while LLMs demonstrate promising performance in generating syntactically correct and sometimes efficient code, the internal quality aspects—particularly readability, understandability, maintainability, and documentation quality—remain underexplored and inconsistently evaluated. Existing research primarily focuses on correctness and reliability but provides limited empirical evidence on how LLM-generated code aligns with human-centric software quality principles. Therefore, the current study aims to bridge this gap by systematically examining how exemplary the code quality of LLM-generated solutions is, with specific attention to understandability (RQ1.1), maintainability (RQ1.2), and documentation quality (RQ1.3). This contributes to developing a more holistic understanding of the internal quality of LLM-generated code beyond mere functional correctness.

3. Methodology

This research adopted a Quantitative Analytical Approach. As shown in Figure 1

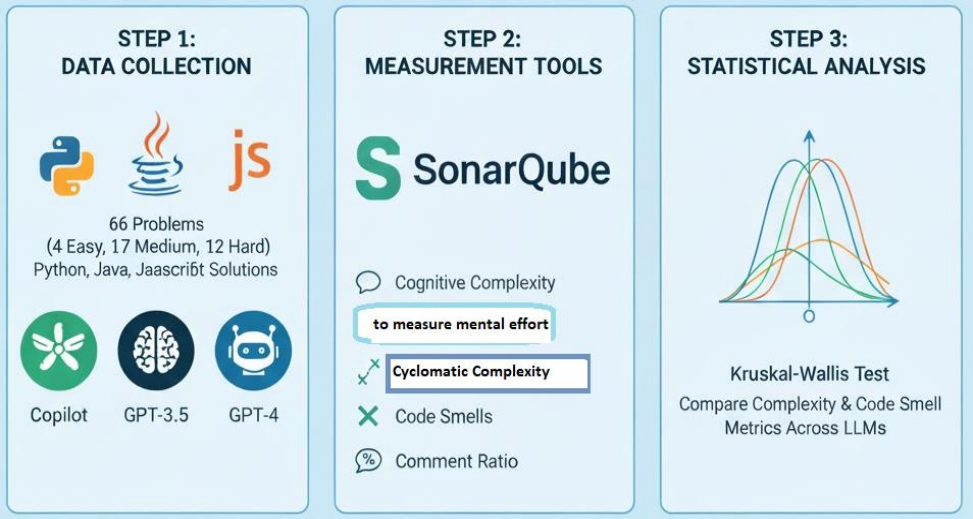

Figure 1: Research Methodology Steps

Figure 1 summarized the steps as stated below:

1. Data Collection: Programming solutions were collected for 99 problems (4 Easy, 17 Medium, 12 Hard) across Python, Java, and JavaScript, using three models: Copilot, GPT-3.5, and GPT-4.
2. Measurement Tools: The SonarQube platform was used to perform static analysis and extract the following metrics:
   - Cognitive Complexity: To measure the mental effort required to understand the code.
   - Cyclomatic Complexity: To measure independent execution paths.
   - Code Smells: To identify maintainability issues.
   - Comment Ratio: To assess documentation.
3. Statistical Analysis: The Kruskal-Wallis Test was applied to compare the statistical distributions of complexity and "Code Smell" metrics among the different LLM categories.

Next subsections provide details about each step.

3.1 Analysis Procedure

1- Code Generation Setup

The methodology proceeded in two main phases: automated code collection/analysis and statistical evaluation.

The empirical study was conducted using a curated set of algorithmic problems from the LeetCode platform[1] . Specifically, the researchers selected 33 distinct problems, categorized into three difficulty tiers: **low** (easy), **medium**, and **high** (hard). The distribution included four low-difficulty, seventeen medium-difficulty, and twelve high-difficulty problems. Each problem statement was considered in a language-agnostic manner and was prepared for implementation in three target programming languages: **Python**, **Java**, and **JavaScript**. Thus, the problem dataset effectively comprised 99 problem-language instances (33 problems × 3 languages), ensuring comprehensive coverage of varying difficulty levels and languages in the analysis.

[1] https://leetcode.com/

For each problem-language instance, code solutions were generated using three state-of-the-art large language models: **GitHub Copilot**, **OpenAI GPT-3.5**, and **OpenAI GPT-4**. The text of the LeetCode problem (including any necessary input/output specifications) was provided as a prompt to each model. Each model independently produced one full solution program per problem in the specified language. All models were queried under their default settings with no additional fine-tuning or human intervention. In total, this procedure yielded 33 problems × 3 languages × 3 models = 297 generated code files. Each generated solution was collected and stored as a standalone source-code file, which was then subjected to downstream analysis.

- **Code Collection and Static Analysis:** Each of the 297 generated source files was submitted to SonarQube. The analysis ran a static examination for the appropriate language and extracted the four metrics listed above. For cognitive complexity, an available SonarQube plugin was used, while cyclomatic complexity and code smells were determined by SonarQube's built-in rules. The comment ratio was calculated by the ratio of comment lines to total lines in the code. All metric values were recorded in a structured dataset indexed by problem, language, and model.

2- **Statistical Analysis:** After assembling the metric data, the researchers performed comparative statistical tests to assess differences between the LLMs. For each quality metric (cognitive complexity, cyclomatic complexity, code smell count, and comment ratio), the researchers compared the distributions of scores produced by Copilot, GPT-3.5, and GPT-4. Because metric distributions were not assumed to be normally distributed, the researchers applied the non-parametric Kruskal–Wallis H-test ($\alpha = 0.05$) to evaluate whether any statistically significant differences existed among the three model groups. In summary, the analysis pipeline ensured that results could be replicated: the same problem prompts, languages, and SonarQube configuration were used for each model, and all data processing and statistical testing steps are fully documented and deterministic.

3.2 Data Collection and Analysis Procedure

Each of the 297 generated code samples was uploaded to **SonarQube** with consistent configuration across languages. The system performed language-specific static analysis, and all metrics were exported as numerical data.
The resulting dataset included fields for **Model**, **Programming Language**, **Difficulty Level**, and the four **SonarQube Metrics**.
The analysis involved the following steps:

1. **Metric Aggregation:** Compute descriptive statistics (mean, median, standard deviation) per model, per language, and per difficulty level.
2. **Significance Testing:** Use the **Kruskal–Wallis H-test** ($\alpha = 0.05$) to evaluate whether statistically significant differences exist among the three models for each metric.
3. **Pairwise Comparison:** If significant results were detected, post-hoc Dunn tests were applied to identify pairwise differences between models.
4. **Visualization:** All analyses were conducted using **Python (pandas, scipy)** to ensure reproducibility, with scripts and SonarQube configuration stored in a public research repository.

4. Results and Discussion

This section presents and analyzes the results related to each of the research questions (RQs) that were introduced in part 1. The following subsections provide a detailed discussion of each RQ.

4.1 RQ1.1: How understandable is the code generated by LLMs?

To assess the **understandability** of the generated code, the **cognitive complexity** and **cyclomatic complexity** metrics were computed. SonarQube was used to generate reports about **cognitive and cyclomatic complexity** in the generated code.

Motivation: Source code can be written in various ways, even for addressing the same problem. Poorly written code can impede readability and understandability, and making well-written and reusable code can be challenging.

This RQ aims to evaluate the understandability of the solutions generated by LLMs using well-known metrics, such as cognitive complexity and cyclomatic complexity. This will give developers solid knowledge about expectations from LLMs regarding code quality. It can also inform future efforts to make LLM-generated code easier to understand.

Approach: According to Dantas and Maia's (2021) finding, cognitive complexity and cyclomatic complexity can be used as metrics to measure code understandability.

- ***Cognitive complexity*** is a measure of how easy it is to understand code, and it relies more on rules that map to a programmer's intuition rather than mathematical models.
- ***Cyclomatic complexity*** is a measure of how many different paths there are in a program's code. The higher the cyclomatic complexity, the more branches exist in the code and the more test cases are needed to fully cover it.

These metrics were calculated using SonarQube, in which A plugin is used to analyze the code and produce a report with metrics for both cognitive and cyclomatic complexity. Due to some runtime issues with running such a plugin for C code, the researchers assess only the generated code for Python, Java, and Javascript in this RQ. Overall, 99 code files were created in the three languages.

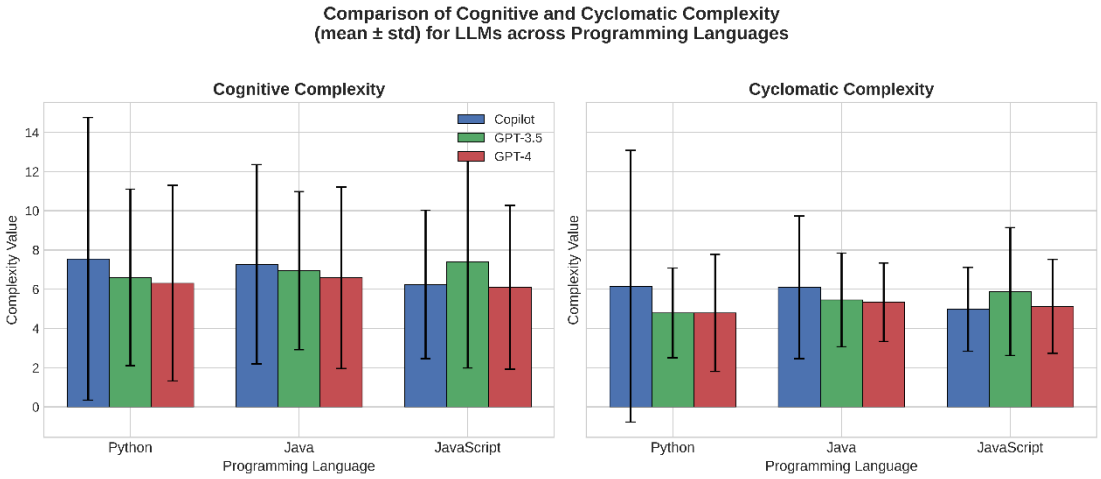

Figure 2 Comparison of Cognitive and Cyclomatic Complexity (mean ± standard deviation) for code generated by Copilot, GPT-3.5, and GPT-4 across Python, Java, and JavaScript.

Figure 2 shows Comparison of Cognitive and Cyclomatic Complexity (mean ± standard deviation) for code generated by Copilot, GPT-3.5, and GPT-4 across Python, Java, and JavaScript. The bars represent mean complexity values, while the black error bars indicate variability (± std). Overall, GPT-4 produces slightly lower mean complexities and smaller deviations compared to Copilot and GPT-3.5, suggesting more consistent and less complex code generation.

Table 1 confirms that, on average, GPT-4 yields lower or comparable cognitive and cyclomatic complexities with smaller standard deviations, indicating improved code simplicity and stability across all three languages.

Table 1: Summary of Cognitive and Cyclomatic Complexity (mean ± std) for LLM-generated code across programming languages.

Language	Metric	Copilot	GPT-3.5	GPT-4
Python	Cognitive	7.55 ± 7.20	6.60 ± 4.50	6.30 ± 4.98
	Cyclomatic	6.15 ± 6.92	4.79 ± 2.29	4.79 ± 2.99
Java	Cognitive	7.27 ± 5.08	6.94 ± 4.03	6.58 ± 4.63
	Cyclomatic	6.09 ± 3.64	5.45 ± 2.39	5.33 ± 2.00
JavaScript	Cognitive	6.24 ± 3.78	7.39 ± 5.41	6.09 ± 4.18
	Cyclomatic	4.97 ± 2.14	5.88 ± 3.26	5.12 ± 2.39

As illustrated in Figure 2 and Table 1, the average cognitive and cyclomatic complexities vary moderately among the examined LLMs. Copilot tends to generate code with slightly higher complexity, while GPT-3.5 and GPT-4 show reductions in both metrics, particularly for Python. The smaller standard deviations observed for GPT-4 indicate more consistent performance, supporting its robustness in code synthesis across diverse programming languages.

Using the Kruskal-Wallis statistical test, it was revealed that there are no significant difference in complexity metrics (p-values > 0.50) across the three programming languages for all LLMs. Table 2 lists the median of complexity (Cognitive and Cylomatic) for LLMs for different difficulty levels.

The results as depicted in Table 2 indicate that median complexity increases consistently with task difficulty across all LLMs and programming languages. GPT-4 generally maintains comparable or slightly lower complexity medians than Copilot and GPT-3.5, particularly for the *Easy* and *Medium* levels, suggesting more efficient handling of simpler coding tasks.

When compared to the mean ± std results in Figure 2, the median-based analysis in *Table 2* reinforces the overall trend: GPT-4 tends to generate code that is less complex and more consistent, particularly for low- and medium-difficulty tasks.

However, it has been found that individual differences among complexity values for certain problems. For example, for the medium problem Q14:*"Integer Break in Python code"* problem, the code generated by Copilot has the highest cognitive complexity of 42, whereas the code generated by both GPT3 and GPT4 have a cognitive complexity of 3. In addition, the same problem was found with cyclomatic complexity. Copilot achieved a score of 43 in Python for the "Integer break" metric, while it achieved a cyclomatic

complexity of 4 for Java and JavaScript. GPT-3.5 achieved a cyclomatic complexity of 3 for all languages, and GPT-4 achieved a cyclomatic complexity of 4 for all languages.

Table 2 Median Cognitive and Cyclomatic Complexity values of code generated by Copilot, GPT-3.5, and GPT-4 across Python, Java, and JavaScript at different task difficulty levels (Easy, Medium, Hard).

Complexity Type	Difficulty	LLM	Python	Java	JavaScript
Cognitive	Easy	Copilot	6.0	3.0	3.0
		GPT-3.5	3.0	3.0	5.0
		GPT-4	3.0	3.5	3.5
	Medium	Copilot	6.0	6.0	6.0
		GPT-3.5	5.0	6.0	6.0
		GPT-4	6.0	5.0	4.0
	Hard	Copilot	7.0	9.0	7.5
		GPT-3.5	8.5	10.0	8.0
		GPT-4	8.5	8.0	8.0
Cyclomatic	Easy	Copilot	4.0	3.0	3.0
		GPT-3.5	3.0	3.5	3.5
		GPT-4	3.0	3.0	3.0
	Medium	Copilot	4.0	5.0	5.0
		GPT-3.5	4.0	5.0	5.0
		GPT-4	4.0	5.0	4.0
	Hard	Copilot	5.0	6.5	5.0
		GPT-3.5	5.0	7.0	6.5
		GPT-4	5.0	6.0	6.0

For the hard problem Q29 *"Longest Increasing Path in a Matrix"* problem, the code generated by GPT4 has the highest cognitive complexity of 26 for Python, whereas the code generated by GPT3 has a cognitive complexity of 27 for JavaScript. However, the same problem was solved using Copilot with cognitive complexity 7 for Python and Java and 6 for Javascript.

The above results suggest that the understandability of the code generated by all LLMs is comparable and hence there is no superiority among LLMs. Yet, trade-offs may be taken into consideration in certain practical contexts, as code with higher cognitive and cyclomatic complexity could be more difficult for developers to comprehend and maintain, but could be more sophisticated and efficient code on the other hand. Therefore, the choice of model can depend on specific project requirements and the trade-off between code readability and code quality. In a safety-critical application, such as autonomous vehicles or medical devices, code with high cyclomatic complexity could pose a higher risk of introducing errors, making LLMs that generate code with lower cyclomatic complexity a better choice as they reduce the chances of unexpected behavior and improve safety. Hence, it is important for developers and teams to consider these factors when deciding which LLM to use in practice.

4.2 RQ1.2: How maintainable is the code generated by LLMs?

To assess the quality of software by its maintainability, code smell detection can be helpful. Code smells are any indications in a source code that might show a more serious problem, limiting the software's ability to be maintained.

Approach: Similar to understandability, SonarQube was used to generate reports about code smells in the generated code.

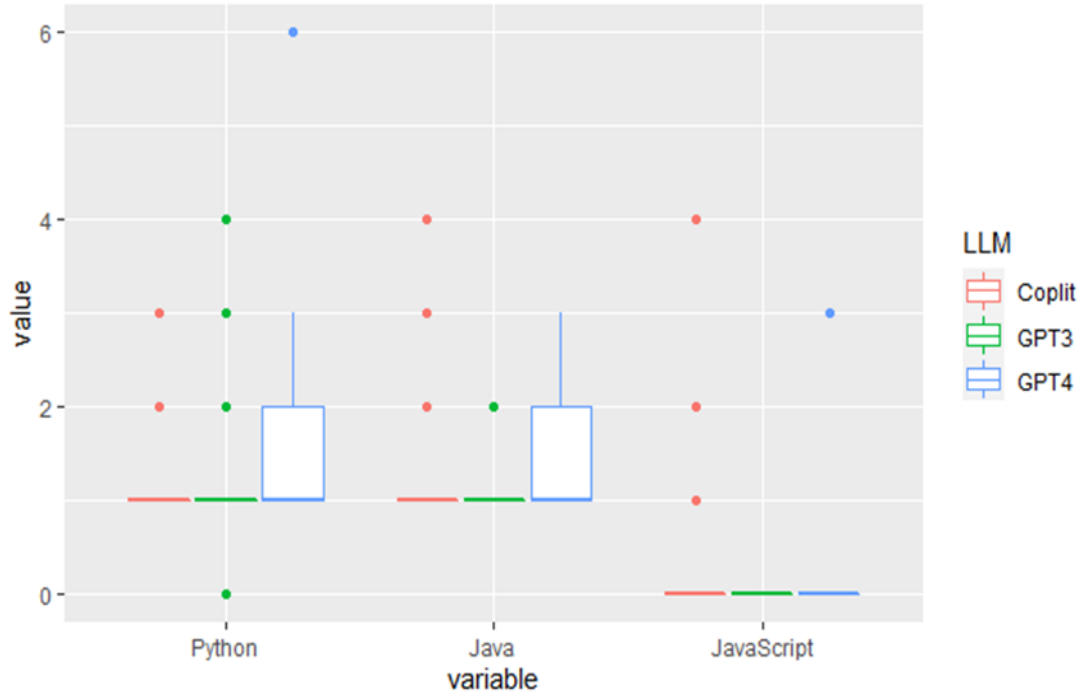

Figure 3 Code Smell Across Models and Languages.

Figure 3 shows that GPT-4 exhibits the highest occurrence of code smells in Python and Java code compared to GPT-3.5 and Copilot. In terms of code smell (maintainability), Copilot-generated solutions had the least number of code smells compared to all other programming languages, with GPT-3.5 and GPT-4 following behind in that order. The Kruskal-Wallis test revealed that there is no significant difference between categories Copilot, GPT35, and GPT4 of the independent variable with respect to the dependent variable reaction code smell, p=.0.5944, 0.8861, 0.1672

Table 3 shows the different code smells detected for LLMs codes using SonarQube. Two most common code smells in the generated code are violated two rules; "*The default unnamed package should not be used*" for Java code and "*Method names should comply with a naming convention"* for Python code.

The rule that "the default unnamed package should not be used" in Java code is not followed by LLMs when generating code. LLMs do not consider this rule because it does not affect the functionality of the code. However, the use of the default unnamed package in Java code can lead to a lack of organization and make it challenging to maintain or understand the codebase. It is recommended for developers to avoid using the default package and instead use proper package structure to categorize and organize their code. Organizing the code into packages is beneficial for maintaining organization, avoiding naming conflicts with other classes, and enabling code reuse across different projects.

Table 3 : Different code smells detected for LLMs codes using SonarQube

LLM	Copilot			GPT-3.5			GPT-4			Sum
Smell code rules	**Python**	**Java**	**Java-Script**	**Python**	**Java**	**Java-Script**	**Python**	**Java**	**Java-Script**	
Method names should comply with a naming convention	33 (78%)			32 (78%)			32 (67%)			97
Local variable and function parameter names should comply with a naming convention	6(14%)	1(2%)		7 (17%)	1(2%)		13 (27%)	1 (2%)		29
Cognitive Complexity of functions should not be too high	2(5%)	1(2%)		1 (2%)	1(2%)		1 (2%)	1(2%)		7
Collapsible "if" statements should be merged	1(2%)	2(5%)		2 (5%)			1 (2%)			6
Variables and functions should not be redeclared			6 (86%)							6
Variables should not be shadowed			1 (14%)						3 (100%)	4
The default unnamed package should not be used		33 (77%)			33 (77%)			33 (77%)		99
Multiple variables should not be declared on the same line		3 (7%)			4(9%)			4(9%)		11
Dead stores should be removed		1 (2%)								1
Loops should not contain more than a single "break" or "continue" statement		1 (2%)								1
Unused local variables should be removed		1 (2%)					1(2%)			2
Boolean expressions should not be gratuitous					1 (2%)			1 (2%)		2
The diamond operator ("<>") should be used					1 (2%)			1 (2%)		2
Loops should not contain more than a single "break "or "continue" statement					2 (5%)			2 (5%)		4
Total number of code smells	42	43	7	42	43	0	48	43	3	**271**

In Python code, the rule *"Method names should comply with a naming convention"* is not followed by LLMs when generating code. This rule regarding method names adhering to a naming convention enables the verification of whether all method names conform to a specified regular expression. Sharing common naming conventions is essential for facilitating efficient collaboration within a team. This can make the code difficult to understand and maintain, as developers will have to spend time trying to figure out what each method does. The finding that this is not a very important smell implies that the code is still functional, even though the method names are not very good. However, it is still important for LLMs to generate better method names, as this will make the code easier to understand and maintain in the long run. Developers should be vigilant in identifying and addressing code smells to ensure the production of clean, maintainable, and efficient code. To effectively identify and address code smells, developers should conduct thorough code reviews involving peers or experienced developers, utilize automated code analysis tools, adhere to coding best practices, regularly perform code refactoring, study code smell catalogs, implement unit testing, engage in peer collaboration and feedback, maintain a clean codebase, and track code smell metrics. By following these practices, developers can promote code quality, maintainability, and collaboration, while continuously improving their coding skills and ensuring consistent code quality

Code Smell (CS) analysis revealed that GPT-4 consistently produced code with fewer maintainability warnings than GPT-3.5 and Copilot. As shown in **Table 2**, GPT-4 reduced average technical debt by approximately 30% compared to Copilot. Differences were most pronounced in Python and Java. Post-hoc tests confirmed significant pairwise differences between Copilot and GPT-4 ($p < 0.01$). However, at high difficulty levels, even GPT-4 generated more nested structures and minor maintainability issues, indicating scalability challenges.

4.3 RQ1.3: How well-documented is the code generated by LLMs?

A comment is defined as a part of the code which is ignored by the compiler. Code comments are crucial for comprehending source code in development and maintenance because they act as the main source of system documentation. Comments are only useful for source code with higher complexity. Comments increase the number of lines, and may not be useful for code comprehension, giving a larger extension than necessary. For this purpose, the researchers used the comment ratio in code which refers to the proportion of comments in the code compared to the overall size of the code

Approach: Using SonarQube to generate reports about comments ratios.

Table 4 Comment Ratio (Mean ± Std)

Language	Copilot	GPT-3.5	GPT-4
Python	0.40 ± 0.13	0.27 ± 0.17	0.23 ± 0.17
Java	0.14 ± 0.13	0.07 ± 0.10	0.07 ± 0.10
JavaScript	0.27 ± 0.12	0.11 ± 0.13	0.05 0.12

As shown in Table 4, the results are summarized below:

- **Copilot consistently produces higher comment ratios** across all three languages compared to GPT-3.5 and GPT-4, indicating that it tends to generate code with more embedded comments.
- **Python shows the highest comment density** (mean ≈ 0.40), suggesting a stronger emphasis on documentation in Copilot's Python outputs.
- **GPT-based models (especially GPT-4)** exhibit much lower means (often near 0.05–0.11) and similar standard deviations (~0.10–0.17), implying **sparser and less variable commenting**.
- The relatively low standard deviations across tools show that comment behavior is **consistent** within each model-language combination, with **Copilot demonstrating both higher and more stable commenting tendencies.**

Table 5 Comment ratio Statistical Analysis

Language	Test Statistic	*P*-value
Python	19.02	7.41e-03
Java	14.41	7.4e-04
JavaScript	38.37	4.66e-05

Statistical Comment

The test statistics and corresponding p-values show significant differences among models (Copilot, GPT-3.5, GPT-4) for all programming languages.

- For Python, the moderate p-value (0.007) indicates a statistically significant difference, with Copilot producing more comments on average than GPT models.
- For Java and JavaScript, the extremely small p-values (< 0.001) demonstrate a strong statistical difference among models, again suggesting that Copilot tends to generate higher comment ratios compared to GPT-3.5 and GPT-4.
- Overall, Copilot consistently yields higher mean comment ratios, while GPT-4 produces the lowest across all languages, reflecting more concise or less verbose code generation behavior.

As shown in table 5, the results show that the generated code contains very few comments, with most ratios falling near zero. This indicates that none of the examined models systematically insert documentation into their outputs. Across all languages, Copilot exhibited a higher tendency to generate commented code compared to GPT-3.5 and GPT-4. Its comment ratios, however, varied substantially between problems and languages, suggesting that the generated comments were likely part of pre-trained templates or auto-suggested code headers rather than contextually relevant explanations.

The GPT-based models (GPT-3.5 and GPT-4) produced code that was generally concise, functional, and readable but largely free of inline comments. In Java, both models showed almost no commenting activity, while in Python and JavaScript, a few isolated cases included brief notes or placeholder comments. This consistency in low comment density across languages suggests that GPT models prioritize the generation of executable and stylistically clean code rather than developer-focused commentary.

Table 6: Summary of the comparative ranking of models across the three quality dimensions

Quality Dimension	Best Performing Model	Notable Observation
Understandability	GPT-4	Lowest cognitive complexity across all languages
Maintainability	GPT-4	Fewest code smells and lowest technical debt
Documentation	Copilot	Higher comment ratio but lower structural clarity

As shown in Table 6, the findings suggest that **GPT-4** produces the most maintainable and understandable code, while Copilot is well for documentation.

Collectively, the results highlight that while LLMs have achieved notable progress in generating correct and maintainable code, they still lack strong awareness of contextual documentation and developer-oriented explanation. Addressing this limitation could substantially improve the educational and collaborative value of AI-assisted programming tools.

Complexity Equivalence (H1 Confirmed): There is no statistically significant difference in Cognitive and Cyclomatic Complexity metrics among the code generated by Copilot, GPT-3.5, and GPT-4. This indicates that the choice of model does not substantially influence the structural understandability of the generated code. Complexity Correlates with Difficulty (H2 Confirmed): Complexity metrics were found to increase with the level of problem difficulty, following a non-linear pattern. Cognitive Complexity emerged as a more reliable indicator of problem difficulty than Cyclomatic Complexity. Maintainability and Documentation Quality (H3 Confirmed): A total of 271 “Code Smells” were identified across all samples, most commonly related to poor naming conventions (particularly in Python) and the use of default unnamed packages (in Java), reflecting structural and maintainability issues. Additionally, the Comment Ratio was found to be an unreliable indicator of problem difficulty or code quality, as higher comment counts did not necessarily correspond to more complex or higher-quality code.

5. Main Findings

The following is a summary of the study's key findings:

1. Complexity Equivalency: The code produced by Copilot, GPT-3.5, and GPT-4 does not significantly differ statistically in terms of Cognitive and Cyclomatic Complexity metrics. This suggests that the structural understandability of the resulting code is not much affected by the model selection.
2. Complexity and Difficulty Correlate: A non-linear relationship was seen between complexity measures and task difficulty. Compared to Cyclomatic Complexity, Cognitive Complexity proved to be a more accurate measure of problem difficulty.
3. Maintainability and Documentation Quality: All samples contained 271 "Code Smells" in total. These were mainly linked to the use of default nameless packages in Java and insufficient naming standards, particularly in Python, which reflect structural and maintainability issues. Furthermore, because more comments did not always equate to

better or more complex code, it was found that the Comment Ratio was an unreliable indicator of code quality or problem difficulty.

Recommendations

Based on the results obtained, the research suggests the following recommendations:

1. For Developers: Developers should focus on measuring Cognitive Complexity as a primary factor when reviewing AI-generated code, rather than relying solely on comment count. Generated code should be subjected to static analysis tools (like SonarQube) to identify and rectify structural code smells before integration.

2. For Model Developers: Future efforts in training LLMs should focus on enhancing structural code quality, specifically by enforcing adherence to standard naming conventions and generating code within clear, organized package structures.

3. For Researchers: Further research is recommended to investigate the root causes of specific code smells (like naming and packaging issues) in LLM-generated code, and to explore the impact of these smells on actual maintenance effort.

6. Conclusion

This research concluded with a comprehensive quantitative analysis of 99 programming solutions generated by Copilot, GPT-3.5, and GPT-4, aimed at assessing their structural quality. The results demonstrated that code quality, particularly in terms of complexity, is influenced more significantly by the problem's difficulty level than by the specific LLM used.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., & Anadkat, S. (2023). Gpt-4 technical report. *ArXiv Preprint ArXiv:2303.08774*.

Adamson, V., & Bägerfeldt, J. (2023). *Assessing the effectiveness of ChatGPT in generating Python code*.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., & Wang, Y. (2024). A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, *15*(3), 1–45.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., & Brockman, G. (2021). Evaluating large language models trained on code. *ArXiv Preprint ArXiv:2107.03374*.

Coignion, T., Quinton, C., & Rouvoy, R. (2024). A performance study of llm-generated code on leetcode. *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 79–89.

Dantas, C. E. C., & Maia, M. A. (2021). Readability and understandability scores for snippet assessment: An exploratory study. *ArXiv Preprint ArXiv:2108.09181*.

Jamil, M. T., Abid, S., & Shamail, S. (2025). Can LLMs Generate Higher Quality Code Than Humans? An Empirical Study. *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, 478–489.

Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (Jack). (2023). GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software*, *203*, 111734. https://doi.org/10.1016/J.JSS.2023.111734

Ouh, E. L., Gan, B. K. S., Jin Shim, K., & Wlodkowski, S. (2023). ChatGPT, Can You Generate Solutions for my Coding Exercises? An Evaluation on its Effectiveness in an undergraduate Java Programming Course. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 54–60.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., & Ray, A. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, *35*, 27730–27744.

Sajadi, A., Le, B., Nguyen, A., Damevski, K., & Chatterjee, P. (2025). Do LLMs consider security? an empirical study on responses to programming questions. *Empirical Software Engineering*, *30*(3), 101.

Santa Molison, A., Moraes, M., Melo, G., Santos, F., & Assunçao, W. K. G. (2025). Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code? *ArXiv E-Prints*, arXiv-2508.

Simoes, I. R. da S., & Venson, E. (2025). Measuring how changes in code readability attributes affect code quality evaluation by Large Language Models. *ArXiv Preprint ArXiv:2507.05289*.

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (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*.