future internet

MDPI

Article

Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation

Davide Tosi

Department of Theoretical and Applied Sciences, University of Insubria, 21100 Varese, Italy; davide.tosi@uninsubria.it

Abstract: The advent of Generative Artificial Intelligence is opening essential questions about whether and when AI will replace human abilities in accomplishing everyday tasks. This issue is particularly true in the domain of software development, where generative AI seems to have strong skills in solving coding problems and generating software source code. In this paper, an empirical evaluation of AI-generated source code is performed: three complex coding problems (selected from the exams for the Java Programming course at the University of Insubria) are prompted to three different Large Language Model (LLM) Engines, and the generated code is evaluated in its correctness and quality by means of human-implemented test suites and quality metrics. The experimentation shows that the three evaluated LLM engines are able to solve the three exams but with the constant supervision of software experts in performing these tasks. Currently, LLM engines need human-expert support to produce running code that is of good quality.

Keywords: generative artificial intelligence; source code generation; software quality; software metrics

Citation: Tosi, D. Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation. *Future Internet* **2024**, *16*, 188. https://doi.org/10.3390/fi16060188

Academic Editor: Ivan Serina

Received: 29 April 2024
Revised: 19 May 2024
Accepted: 23 May 2024
Published: 24 May 2024

1. Introduction

The advent and subsequent burgeoning of Artificial Intelligence (AI) is drastically changing numerous fields, ushering innovative possibilities, particularly regarding software development by means of AI generative models capable of automatically generating source code. The emergence of these models and their abilities in software development and code generation is offering the possibility for enhancing productivity, optimization, and the redefinition of development and software engineering practices [1–4]. This paper delves into the evaluation of the capabilities of these models, specifically Large Language Models (LLMs), in generating source code.

The proliferation of AI generative engines and LLMs advances the field of code generation, starting from the training on vast data comprised of code and natural language, thus facilitating the comprehension and generation of human-like text, code, and programming concepts. As these models further integrate into the software development process, concerns regarding the functionality and the quality of the generated code suggest the necessity of assessing these models both from functional and non-functional points of view. To obtain empirical evidence about the abilities of these LLMs in understanding coding problems and in developing the related source code, an empirical validation (through systematic observation, experimentation, and experience in real-world coding problems) has been conducted.

The methodology devised for this study comprises a structured and sequential approach, intended for the thorough evaluation of the capabilities of the tested LLMs in generating functional, quality code. The methodology incorporates both qualitative and quantitative evaluations to compare the performance of the tested LLMs when facing various programming scenarios or coding problems. The validation starts with the identification of six coding problems and the selection of three LLMs to be compared; then, the code generated by each LLM model is evaluated by means of a two-fold approach based

on (1) the execution of a test suite to verify the functional correctness of the generated code, and (2) the quality of the code generated is evaluated through a selection of software quality metrics.

This empirical evaluation of the quality of AI-generated code highlights models' capabilities and limitations in developing software by guiding developers, programmers, and researchers towards the evaluation of the readiness and reliability of these models, and thereby, the decision to integrate and leverage these models in programming, and coding practices.

2. Related Work

Researchers are putting a lot of effort into evaluating different aspects related to the source code generated by LLMs, such as ChatGPT.

In [5], the authors focus on the implementation of EvalPlus [6], a benchmarking framework designed to assess the correctness of LLM-generated code by supplementing a substantial quantity of test cases using an LLM-based and mutation-based (test case) generation approach. EvalPlus can be applied to further extend another benchmarking framework, HUMANEVAL, significantly increasing the quantity of test cases (or evaluation scenarios) compared to just using HUMANEVAL as a standalone, resulting in the benchmarking framework HUMANEVAL+. Through a test-suite reduction, another benchmarking framework, HUMANEVAL+ MINI, can be derived that reduces HUMANEVAL+ test cases while maintaining the same level of effectiveness. As such, the pass@k metric (that indicates the proportion of correct items within the top "k" positions) for HUMANEVAL+ and HUMANEVAL+ MINI lowers, suggesting that the number and quality of test cases can significantly impact the assessment of the correctness of LLM-generated code.

In [7], the authors show that the generated programs (or code) by ChatGPT (GPT-3) [8] fall below the minimum security standard for most contexts. When prodded, GPT-3 recognized and admitted the presence of critical security issues and vulnerabilities in the generated code but was then able to generate the secure version of the code if explicitly asked to do so. GPT-3 further provided explanations about the vulnerability/exploitability of the generated code, offering a pedagogical (instructive) value, or an interactive development tool. Another concern related to the generated code is code secrecy. The code generated by GPT-3 closely resembles confidential company data and information since employees rely on GPT-3 to aid them in writing documents or other managerial tasks. Since the interaction will be registered for the GPT-3's knowledge base, this can expose/leak business/corporate secrets, or confidential information. It is generally accepted that sharing code (i.e., open source code) makes software more robust but opens issues to code secrecy and cybersecurity.

In [9], the evaluation of code generated by ChatGPT (GPT-3) is analyzed from three different aspects: correctness, understandability, and security with a multi-round fixing process. Outputs demonstrate that (1) GPT-3 generates functionally correct code for Bef (before) problems better than for Aft (after) problems, but its ability to fix and correct already faulty code to achieve the desired behavior is relatively weak. (2) The Cognitive and Cyclomatic Complexity levels vary across different programming languages (due to language-specific features, syntax, or the nature of the problems being solved). Also, the multi-round fixing process generally preserves or increases the complexity levels of the code. (3) And the code generated by GPT-3 has vulnerabilities, but the multi-round fixing process demonstrates promising results in successfully addressing vulnerabilities (by complementing GPT-3 with vulnerability detection tools such as CodeQL [10] mitigates code generated with vulnerabilities). GPT-3 is a closed-source model, meaning that the internal engine of GPT-3, or the specific workings of the model, remains unknown, making it difficult to ascertain whether the coding problems have been previously used in the training dataset. The responses generated by GPT-3 only reflect the abilities of the model at the time of writing, but the model is continuously training and evolving. The LeetCode online judgment tests the functional correctness, and CodeQL (with manual analysis)

detects potential vulnerabilities in the generated code; the LeetCode [11] and CodeQL feedback can then be used to prompt ChatGPT again for a new iteration of code generation.

In [12], it is shown that ChatGPT's (GPT-3) performance declines over the difficulty level (i.e., it behaves better on simple tasks than on medium and hard coding problems) and the time period of the code tasks. The engine's ability to generate running code is inversely proportional to the size of the generated code, thus indicating that an increased complexity of these coding problems raises important challenges for the model. The generated code is also prone to various code quality issues, compilation and runtime errors, wrong outputs, and maintainability problems; the overall performance is directly influenced by the coding problem's difficulty, task-established time, and program size, and the ability to overcome these issues depends on the type of human feedback prompted, the programming language, and the specific quality issue under analysis. For example, static analysis feedback works well for code style and maintainability issues, while simple feedback is more effective for addressing functional errors.

In a comparative study of GitHub Copilot [13], Amazon CodeWhisperer [14], and ChatGPT (GPT-3) on 164 coding problems [15], GitHub Copilot achieved a 91.5% success rate (150 valid solutions), Amazon CodeWhisperer 90.2% (148 valid solutions), and GPT-3 the highest at 93.3% (153 valid solutions). The main issues leading to invalid code across these tools included operations with incompatible types, syntax errors, and the use of functions from unimported libraries. Amazon CodeWhisperer also faced issues with improper list indexing, searching for non-existent values in lists, incorrect assert statement usage, and stack overflow errors. GPT-3's errors also involved improper list and string indexing. Despite these issues, the study suggests that the performance of these code generation tools is broadly similar, with an average ability to generate valid code 9 out of 10 times. However, it highlights a specific concern regarding operations with incompatible types, which may not always be immediately noticeable to programmers, potentially leading to code failure under different inputs.

In [16], GPTutor is a Visual Studio Code plugin that exploits GPT-3 to generate accurate descriptions of source code. Students can have personalized explanations for coding problems they encounter. Those seeking to learn a new programming language can use GPTutor to navigate several code examples or to quickly familiarize themselves with a code-base to have clarification of the business logic behind each line of code. It uses the OpenAI ChatGPT API to generate detailed explanations of the given source code, giving GPTutor the potential to surpass other code-explaining applications, such as ChatGPT or GitHub Copilot (using the NGL model), with advanced prompt designs.

Ref. [17] discusses the validity threats in an empirical study on ChatGPT's programming capabilities and the steps taken to mitigate them. To combat the inherent randomness in ChatGPT's responses, the study averaged results from multiple queries and used a large dataset for program repair to minimize variability. The selection of benchmarks was carefully considered to avoid data leakage issues. Internally, the study addressed ChatGPT's tendency to mix code with natural language by using scripts and manual checks to ensure only code was evaluated. Annotations in benchmark submissions that could bias code summarization assessments were also removed. Despite using the GPT-3.5 model due to limitations with the newer GPT-4, the study acknowledges that continuous updates by OpenAI might mean that the results underestimate ChatGPT's true capabilities, with plans to explore GPT-4 in future research once it is proven stable. These measures collectively aim to provide a more accurate and reliable evaluation of ChatGPT's potential as a programming assistant.

Ref. [18] shows that GPT-3 exhibits an overall successful rate (across the entire dataset) with the generated solutions. Feedback and error messages from the Leetcode platform, along with failed test cases, guided GPT-3 in improving the generated code. Despite the guidance, GPT-3 struggled to produce correct solutions. Attempts to rectify the errors led to a decrease in performance, with revised solutions failing more test cases. The inability to generate correct solutions and the performance downgrade highlight GPT-

3's limitations in effectively incorporating debugging feedback, hindering the ability to improve solution correctness. Evaluating the success rate of GPT-3 in solving problems across different application domains shows that its highest success rates are with problems coming from "Tree" and "Divide and Conquer" algorithms, contrary to solving problems from "Greedy" and "Dynamic Programming" domains; hence, GPT-3 performs poorly in generating code for problems that follow a well-structured methodology or adhere to established programming patterns, rather than solutions that require complex decision-making processes. In approaching problems of varying difficulties and acceptance rates, the model encounters more challenges with problems labeled as "Hard", rather than the ones labeled as "Easy". Other considerations for the generated solutions are runtime and memory efficiency.

In [19], the authors present an approach for exploring the code generation abilities of GPT-3 through the analysis of crowdsourced data on Twitter (now X) and Reddit. Python and JavaScript are the programming languages most discussed on social networks, and GPT-3 is used for several different tasks, such as preparing programming interviews and solving academic assignments. Sentiment analysis suggests that final users have fears about the code-generation capabilities of ChatGPT. Flake8 [20] is used to assess the quality of the generated code by GPT-3.

In this paper, we focus on code correctness and code quality by using specific test suites and quality metrics to be applied to LLM-generated code.

3. Materials and Methods

3.1. Methodology Overview

The empirical validation conducted in this study encompasses the following phases:

- Selection of 3 LLM engines to be tested and compared;
- Definition of a set of coding problems on different programming concepts and with different difficulties;
- Execution of the 3 LLMs on the selected coding problems and generation of the source code per each coding problem;
- Evaluation of the functional correctness of the generated code by means of a dedicated test suite;
- Evaluation of the code quality through the collection of quality metrics.

This methodology aims to help developers and software engineers in answering the following questions: How effectively do Large Language Models generate functional, quality code with various coding problems? How much do the generated codes align with programming, coding, standards, and practices? How do various Large Language Models compare with their code generation capabilities? What is the comparative quality of code generated by different Large Language Models in terms of readability, security, maintainability, and adherence to standard coding practices?

3.2. Selection of the Large Language Models

The selection process of the 3 LLMs focuses on well-recognized engines with technical capabilities, particularly for problem solving and code generation.

The compilation of suitable LLMs for the study, culminating in a list that, aside from ChatGPT [8] as the primary model, included several other relevant and notable models: GPT-4 [8], Google Bard [21], Microsoft Bing AI Chat [22], Meta Llama [23], Amazon CodeWhisperer [14], GitHub Copilot [13], and Claude [24]. Various requirements were considered and limited the final choice to the following 3 LLMs: ChatGPT with its current GPT-3.5 engine, GPT-4, and Google Bard. We excluded all the other engines with the following motivations (at the time of writing): Microsoft Bing's AI Chat prevented the prompting for code correction since it limits message interactions; Meta's Llama, Amazon's CodeWhisperer, and GitHub Copilot avoid downloading the program with the model; and Claude lacks specific features for code generation. Besides Bard, Google released another

more recent development called Gemini [21], a model boasting advanced features for code generation. However, right now, it is unavailable in Europe.

Hence, the exclusion of these models narrowed down and drew the remaining models for the study: ChatGPT (GPT-3.5), GPT-4, and Google Bard. GPT-4, the latest available release from OpenAI, stands at the forefront of technological advancement for AI and language processing, and is the most advanced and notable model currently available, with capabilities extending significantly from the predecessor, ChatGPT (GPT-3.5); a comparative analysis between these two models offered an opportunity to evaluate the extent of the improvements from ChatGPT (GPT-3.5) to GPT-4, verifying whether real advancements applied to the language processing and code generation. Bard, developed by Google, is the most recognizable competitor to both ChatGPT and GPT-4.

3.3. Definition of the Coding Problems

For the definition of the coding problems, the study firstly leveraged LeetCode [11], a platform renowned for boasting an extensive repository of coding problems that range in diverse programming concepts and difficulties. The problems' complexity and difficulty push the boundaries and thoroughly test the capabilities of the models, mainly to highlight their limitations. The availability of solutions on LeetCode provides the benchmark for comparing the models' solutions and the developers' solutions. Moreover, the clarity and specificity of the statements and descriptions of the LeetCode coding problems avoid ambiguous problem formulation, thus minimizing misunderstandings or misinterpretations that arise from poorly worded or vague problem statements or descriptions.

Three different categories of coding problems from LeetCode were selected to assess the capabilities of ChatGPT (GPT-3.5), GPT-4, and Google Bard in solving these coding problems and to compare their performance from functional and code quality points of view: (1) mathematical, numerical computations and logic; (2) algorithmic, problem-solving design; and (3) data structures manipulation.

Unfortunately, after presenting the selected LeetCode coding problems to each of the selected LLMs for code generation, a consistent, and therefore concerning, pattern emerged from their responses: for all 3 coding problems, ChatGPT (GPT-3.5), GPT-4, and Google Bard returned identical solutions. By comparing the LLMs solutions with the solution provided by LeedCode, we observed that all 3 LLMs retrieved the solutions directly from the LeetCode platform, rather than independently solving the coding problems. This suggests that the LLMs relied on their vast training datasets, which included said solutions, as responses to the coding problems, thus replicating rather than generating the solutions.

Such an implication prompted a reevaluation and redirection regarding the selection of the coding problems for the study, thus shifting the focus to a different source. Firstly, we tried rephrasing the LeetCode problems, but also, in that case, the 3 LLMs were able to find the correct available solutions. Then, we invented 3 simple programming problems, but also, in that case, the 3 LLMs behaved identically. Hence, 3 complex coding exams of the Java Programming Course for the Computer Science degree at the University of Insubria [www.uninsubria.it] were selected, based on the understanding that coding problems from a university programming course would be distinct and unavailable and inaccessible outside the academic setting, being bespoken and specifically designed according to the specific curriculum, thereby reducing the probability of the models replicating available solutions and compelling the models for authentic problem solving and code generation. Here follows the text of the 3 selected coding exams:

Problem 1—The ArrayDifferenza coding problem: write the code for the method

```
public static int[] diff(int[] a, int[] b)
```

which must return the array that contains all and only the elements of array *a* that are not present in array *b*. Assume that the method is always invoked with arguments other than *null*.

Problem 2—The ComparaSequenze coding problem: write the code for the *ComparaSequenze* class that performs as follows:

1. Acquires from the standard input a sequence A of real numbers inserted one after the other (the insertion of numbers ends when the number 0 is inserted);
2. Acquires from the standard input a sequence B of fractions (instances of the Fraction class; some relevant methods of said class: isLesser, isGreater, getNumerator, getDenominator, etc.) inserted one after the other (the insertion of fractions ends when a fraction less than 0 is inserted);
3. Prints on the standard output the fractions in B that are greater than at least half of the real numbers in A.

If at least one of the two sequences is empty, the execution must be interrupted, printing on the standard output an appropriate error message.

Problem 3—The MatriceStringa coding problem: Consider the *MatrixString* class with the following structure:

```
public class MatriceString { private String[] [] m;
...
}
```

1. Write the constructor of the class with the following structure public MatriceStringa(int r, int c, String val), which has the purpose of initializing the field m with a matrix of r rows and c columns in which each position contains the value val. The constructor must throw a RuntimeException if the values r and c are not admissible (utilize an argument-free constructor of the class Runtime Exception).
2. Write the method of the MatrixString class with the prototype public void set(int r, int c, String val) throws MatriceException which assigns to the position of matrix m of row r and column c the value val. The method must throw the unchecked exception MatrixException if the values of r and c are outside the admissible bounds (utilize the constructor without arguments to create the exception).
3. Write the method of the MatrixString class with the prototype public String rigaToString(int idx, String separatore) throws MatriceException which returns the string obtained by the concatenation of the strings that appear in the row of index idx of matrix m separated from each other by the string indicated as separator. The method throws the unchecked exception MatrixException if index idx is not a row of the matrix or if the separator is null.

3.4. LLMs Execution and Code Generation

The coding problems have been submitted to the selected LLMs by maintaining the exact phrasing and language used in the exams in order to avoid any potential linguistic biases that might arise from altering the wording or structure of the coding problems, and that could influence the responses from the models. The generated code has been saved for further analysis.

3.5. Test Suite and Test Cases

The creation of specific test suites per each coding problem is the basis for evaluating the functional correctness and robustness of the generated code. The designed and implemented test cases (TCs) range from basic to complex tests that cover all the coding aspects of each problem to stress the LLMs' strengths and weaknesses. Regarding correctness, we refer to the code performing the intended task and producing accurate outputs for given inputs; as for robustness, we refer to the evaluation through boundary cases and unexpected inputs to determine the code's ability to handle edge cases and potential errors.

In detail, a total of 32 test cases were designed and implemented: 9 TCs for Problem 1, 8 TCs for Problem 2, and 15 TCs for Problem 3. Appendix A lists all the test cases implemented.

3.6. Code Quality Analysis Tool and Selected Metrics

SonarQube [25] and SonarCloud [26] were selected as reference and renowned platforms to collect various metrics for quality, readability, and maintainability. Here is the list of the collected code metrics.

The *Passed Test Cases* metric represents the number of test cases in a test suite that have passed; passing a test case typically means that the program behaves as expected under the conditions defined in that test case.

The *Failed Test Cases* metric represents the number of test cases in a test-suite that have failed; failing a test case typically means that the program behaves not as expected under the conditions defined in that test case, indicating potential bugs, or issues, in the code.

The *Lines of Code (LOC)* metric counts the number of lines in a codebase, indicating the size and complexity of the program; it includes executable code, comments, and blank lines.

The *NumOfMethods* metric represents the total number of methods, or functions, in the codebase, indicating the size and complexity of the program. A high number of methods implies that the code requires refactoring for improved modularity or readability.

The *Cyclomatic Complexity* metric measures the number of linearly independent paths through the program's codebase, used to measure the complexity of a program. A higher Cyclomatic Complexity implies a program with more potential paths, indicating more potential for errors and more effort required for understanding, testing, and maintenance.

The *Cognitive Complexity* metric measures the difficulty in understanding the program or codebase; unlike Cyclomatic Complexity, being based on the program's structure, Cognitive Complexity considers factors like nesting levels, abstraction, and the readability of the code. A higher Cognitive Complexity implies a program that is difficult to understand.

The *Bugs Detected* metric counts the number of issues in the code considered as bugs, identified through testing, or code analysis; bugs are flaws in the code that cause incorrect results or unintended behavior.

The *Vulnerabilities* metric refers to weaknesses in the code that can be exploited, such as security breaches or data leaks; identifying and addressing vulnerabilities ensures security. A higher amount of vulnerabilities implies less secure programs.

The *Hotspots Reviewed* metric indicates sections of code identified as having high potential for errors or complexity, and vulnerable in regards to security; reviewing hotspots ensures maintenance and security.

The *Code Smells* metric indicates potential problems in the code; unlike bugs, Code Smells allow the functioning of the program but indicate weaknesses that might slow down development or increase the risk of bugs or failures in the future.

The *Duplications* metric refers to the presence of duplicated code within the codebase; duplications lead to maintenance being more difficult, indicating that changes might require replication on all duplications and lack of structure and organization.

4. Results

In this section, a summary of the results obtained when executing the test suites on the generated code per each coding problem is listed. Moreover, this section reports on the metrics collected by analyzing the generated source code with the help of SonarCloud. The results are presented as a set of tables that will be discussed in Section 5.

4.1. Results for Problem 1 "ArrayDifferenza"

As for Problem 1, in general, ChatGPT (GPT-3.5) generated functional code on the first try, with no further prompts required. GPT-4 generated non-running code, requiring further prompting to include a *main* method, and then generate running code. Bard generated non-running code, requiring further prompting to include the *import* statements and a *main* method with predefined inputs, and then generate running code. Table 1 lists the result of the execution of the nine test cases implemented for Problem 1. All three LLM engines failed TC6. Table 2 reports on the collected metrics.

Table 1. Passed and failed test cases for Problem 1 "ArrayDifferenza".

	ChatGPT (GPT-3.5)	GPT-4	Bard
Passed Test Cases	8	8	8
Specific Passed Test Cases	*TC1, TC2, TC3, TC4, TC5, TC7, TC8, TC9*	*TC1, TC2, TC3, TC4, TC5, TC7, TC8, TC9*	*TC1, TC2, TC3, TC4, TC5, TC7, TC8, TC9*
Failed Test Cases	1	1	1
Specific Failed Test Cases	*TC6*	*TC6*	*TC6*
Total Test Cases	9	9	9

Table 2. Metric values for Problem 1 "ArrayDifferenza".

	ChatGPT (GPT-3.5)	GPT-4	Bard
Lines of Code (LOC)	30	33	30
Number of Methods (NumOfMethod)	2	2	2
Cyclomatic Complexity	6	8	8
Cognitive Complexity	5	10	10
Bugs Detected	0	0	0
Vulnerabilities	0	0	0
Hotspots Reviewed	0	0	0
Code Smells	4	3	3
Duplications	0.00%	0.00%	0.00%

4.2. Results for Problem 2 "ComparaSequenze"

As for Problem 2, in general, ChatGPT (GPT-3.5) generated non-running code on the first try, requiring further prompting to correct *errors* and adjusting the methods in the *Frazione* class, and then generate functional code. GPT-4 generated non-running code, requiring further prompting to adjust the *methods*, and then generate running code. Bard generated non-running code, requiring further prompting. Bard specified the requirement of a separate class, but even with further prompting, the model failed to generate "functional" code. ChatGPT (GPT-3.5) and GPT-4 were able to create separated classes to solve Problem 2, while Bard failed at the creation of separate classes, compared to the other models. Table 3 lists the result of the execution of the seven test cases implemented for Problem 2. Both ChatGPT (GPT-3.5) and GPT-4 failed TC4. Table 4 reports on the collected metrics.

Table 3. Passed and failed test cases for Problem 2 "ComparaSequenze". Note: Bard is not able to solve the problem.

	ChatGPT (GPT-3.5)	GPT-4	Bard
Passed Test Cases	7	7	/
Specific Passed Test Cases	*TC1, TC2, TC3, TC5, TC6, TC7, TC8*	*TC1, TC2, TC3, TC5, TC6, TC7, TC8*	/
Failed Test Cases	1	1	/
Specific Failed Test Cases	*TC4*	*TC4*	/
Total Test Cases	8	8	/

Table 4. Metric values for Problem 2 "ComparaSequenze".

	ChatGPT (GPT-3.5)	GPT-4	Bard
Lines of Code (LOC)	68	48	/
Number of Methods (NumOfMethod)	6	3	/
Cyclomatic Complexity	15	11	/
Cognitive Complexity	13	11	/
Bugs Detected	0	0	/
Vulnerabilities	0	0	/
Hotspots Reviewed	0	0	/
Code Smells	9	6	/
Duplications	0.00%	0.00%	/

4.3. Results for Problem 3 "MatriceStringa"

As for Problem 3, in general, ChatGPT (GPT-3.5) generated non-running code on the first try, requiring further prompting to include a *main* method with predefined values and generate functional code. GPT-4 generated non-running code, requiring further prompting to include a *main* method with predefined values, and generate functional code. Bardgenerated non-functional code, requiring further prompting to include a *main* method with predefined values and correct *errors* and generate functional code. Table 5 lists the result of the execution of the 15 test cases implemented for Problem 3. ChatGPT (GPT-3.5) failed 2 TCs out of 15 (TC6, TC7), while GPT-4 and Bard failed TC6. Table 6 reports on the collected metrics.

Table 5. Passed and failed test cases for Problem 3 "MatriceStringa".

	ChatGPT (GPT-3.5)	GPT-4	Bard
Passed Test Cases	13	14	14
Specific Passed Test Cases	*TC1, TC2, TC3, TC4, TC5, TC8, TC9, TC10, TC11, TC12, TC13, TC14, TC15*	*TC1, TC2, TC3, TC4, TC5, TC7, TC8, TC9, TC10, TC11, TC12, TC13, TC14, TC15*	*TC1, TC2, TC3, TC4, TC5, TC7, TC8, TC9, TC10, TC11, TC12, TC13, TC14, TC15*
Failed Test Cases	2	1	1
Specific Failed Test Cases	*TC6, TC7*	*TC6*	*TC6*
Total Test Cases	15	15	15

Table 6. Metric values for Problem 3 "MatriceStringa".

	ChatGPT (GPT-3.5)	GPT-4	Bard
Lines of Code (LOC)	64	57	60
Number of Methods (NumOfMethod)	5	4	4
Cyclomatic Complexity	19	20	16
Cognitive Complexity	14	18	13
Bugs Detected	0	0	0
Vulnerabilities	0	0	0
Hotspots Reviewed	0	0	0
Code Smells	4	3	3
Duplications	0.00%	0.00%	0.00%

5. Discussion

The aggregation of the results discussed in Section 4 shows that ChatGPT (GPT-3.5) passed 28 TCs and failed 4 TCs, GPT-4 passed 29 TCs and failed 3 TCs, while Google Bard passed 22 TCs and failed 10 TCs (e.g., we considered all the TCs related to Problem 2 to have failed, where Bard was not able to provide a running code). Figure 1 shows a histogram with the overall distribution of passed/failed TCs for the three LLM engines. Table 7 summarizes the aggregated value of each metric for the three considered coding problems. Also, in this case, it is important to remember that Bard's values do not include metrics for Problem 2.

As for Problem 1, the three LLM engines behaved in a very similar way: similar LOCs, number of methods, a Cyclomatic Complexity near 8 (and in any case lesser than the critical value of 11 that indicates a moderate risk and complexity of the code); a Cognitive Complexity lesser than the critical score of 15; and very few detected potential Code Smells.

As for Problem 2, it is clear that GPT-4 performed better than GPT-3.5, with a more concise code (48 LOCs vs. 68 LOCs, and three methods instead of six methods), and less complex code for the Cyclomatic, Cognitive, and Code Smells metrics.

As for Problem 3, the three LLM engines behaved in a similar way: GPT-4 was able to produce a running solution with fewer LOCs. However, all three LLM engines provided complex running solutions with Cyclomatic and Cognitive Complexities above the recommended thresholds. Few potential Code Smells were detected. For this problem, it should be of interest to ask the three LLM engines to refactor the code in order to reduce the code complexity.

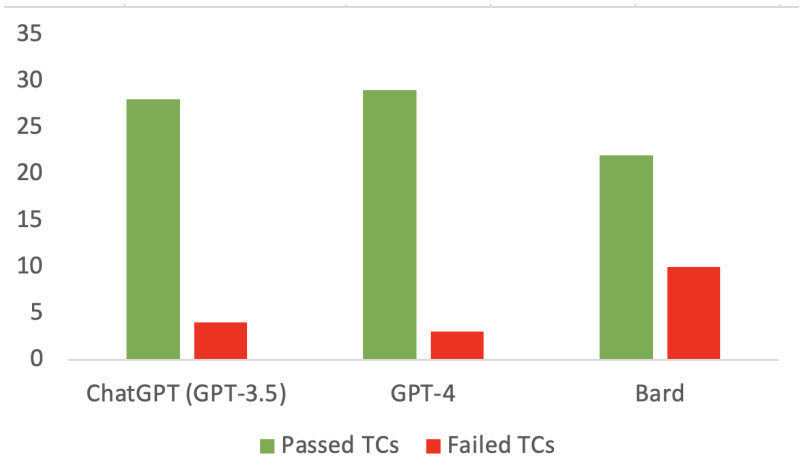

Figure 1. Total passed/failed test cases for all the coding problems.

Table 7. Aggregated metrics values for all 3 coding problems.

	ChatGPT (GPT-3.5)	GPT-4	Bard
Lines of Code (LOC)	162	138	90
Number of Methods (NumOfMethod)	13	9	6
Cyclomatic Complexity	40	39	24
Cognitive Complexity	32	39	23
Bugs Detected	0	0	0
Vulnerabilities	0	0	0
Hotspots Reviewed	0	0	0
Code Smells	17	12	6
Duplications	0.00%	0.00%	0.00%

In general, prompting the coding problems to the various LLMs resulted in generating "testable" code, or viable code for further evaluation, requiring additional, albeit minimal, prompting to generate solutions with no errors for the three coding problems; these prompts served to correct errors, refining the solutions to a point where the generated code became functional, ensuring that the resultant code could be assessed using test cases and metrics, and allowing the realization of the study. Such interactive prompting highlights the dynamic behavior of the models but also the need for human-expert intervention to support and complement the work of LLM engines.

Regarding the performance with designated test cases of various engines, they exhibited a high success rate, effectively passing a majority of the test cases designed to evaluate the functionalities of the generated code, with minimal instances of failure, typically restricted to one or two test cases, suggesting that all the various models generated functional code. Such a high success rate across various models reflects the robustness and reliability of LLMs in code generation and their potential utility in practical programming and software development; the failures do not significantly detract from the overall functionality of the code generated by these models. This also suggests the need to have expert SW testers who monitor the functional correctness of the LLM generated code.

Deriving the coding problems from a university programming exam resulted in concise solutions, reflected by the few lines of code and number of methods within each solution; such brevity seems typical for coding problems designed for academic assessments, with a focus on algorithmic concepts rather than extensive practices. Evaluating the Cyclomatic and Cognitive Complexity, this solution tended towards moderate complexity, indicating that despite their brevity, the solutions entail a certain level of intricacy; such complexity suggests that understanding and maintaining the code requires a more significant effort, possibly necessitating code refactoring. Also, the presence of Code Smells underscores the importance of inspection, even when working with advanced LLMs, demonstrating that the models still necessitate a careful evaluation to align with standards and practices in programming.

As for the rest of the metrics, the code generated by the models for each of the coding problems boasts no bugs detected, vulnerabilities, or duplications, with no need for reviewed hotspots, suggesting high code quality, even if the solutions show moderate complexity and quality concerns; the code demonstrates a commendable level of reliability and security.

6. Threats to Validity

Ensuring the validity of this study is essential for understanding the validity and reliability of this work. For this reason, in this section, we examine potential threats to construct, internal and external validity, aiming to maintain the robustness of our findings.

Construct validity determines whether the implementation of the study aligns with its initial objectives. The efficacy of our search process and the relevance of coding problems and LLM-selected engines are crucial concerns. While our selected coding problems and LLM engines were derived from well-defined research questions and evaluations, the completeness and comprehensiveness of these problems and engines may be subject to limitations. Additionally, the use of different problems and LLM engines might have returned other relevant outcomes and insights that have not been taken into consideration. However, this study highlighted relevant aspects that complement other related work.

Internal validity assesses the extent to which the design and execution of the study minimize systematic errors. A key focus is on the process of test case execution and metrics extraction errors. To minimize this risk, we selected state-of-the-art tools and facilities to conduct test cases and metrics collection in order to minimize errors in the process. Moreover, different researchers can generate different test cases, which can highlight different outcomes. To minimize these risks, two researchers worked separately in designing the test suites, and we merged the resulting test cases in the adopted test suites.

External validity examines the extent to which the observed effects of the study can be applied beyond its scope. In this work, we concentrated on research questions and quality assessments to mitigate the risk of limited generalizability. However, the LLM topic is a very hot topic that changes every day, with new solutions, engine versions, and approaches, thus limiting the external validity and generalizability of findings. Recognizing these constraints, we believe that our work can provide other researchers with a framework that can be used to evaluate different LLM engines, new versions, and different programming languages.

By acknowledging these potential threats to validity, we strive to enhance the credibility and reliability of our work, contributing valuable insights to the evolving landscape of LLM code generation.

7. Conclusions

The exploration into the capabilities of Generative Artificial Intelligence in the field of code generation culminated in a comprehensive analysis, revealing nuanced insights into the performance of LLMs. Through an empirical evaluation, this study navigated the intricate aspect of AI-generated code, scrutinizing the functionality and quality through a series of rigorously designed coding challenges and evaluations.

The results of this empirical evaluation suggest that GPT-3, GPT-4, and Bard can generate the same functional and quality code for coding problems that have available solutions online. As for complex problems, the three evaluated LLMs provided quite similar solutions without bugs, vulnerabilities, code duplication, or security problems detected by SonarCloud. Hence, LLMs try to adhere to standard coding practices when creating source code. However, human supervision is essential to push LLMs in the right direction. Moreover, it is fundamental to prompt LLMs with clear software requirements and well-defined coding problems, otherwise the generated code is too vague or not running (as in the case of Bard for coding problem #2.)

The implications of the study offer insights for programmers, developers, and SW engineers in the field of software development. The evidence presented lays the groundwork for informed decision-making regarding the integration of AI into programming and coding practices, suggesting a constant collaboration between different human experts (such as software engineers, developers, and testers) and AI to elevate the standards of code quality and efficiency.

The study paves the way for future research to explore more LLM engines, diverse coding problems, and advanced evaluation metrics to redefine the concepts of code development that will see strict collaboration among all software stakeholders and AI technologies.

Funding: This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

Data Availability Statement: The data presented in this study are available in this article. Additional source code can be shared up on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MDPI	Multidisciplinary Digital Publishing Institute
DOAJ	Directory of open access journals
TLA	Three-letter acronym
LD	Linear dichroism

Appendix A

Appendix A.1. Test Cases

"ArrayDifferenza" Problem 1 Test Cases:
Test Case 1: Identifies and returns the elements that are unique to the first array when both arrays have common and unique elements.

```
@Test
public void testNormalCase() {
    assertArrayEquals(new int[]{3, 9},
    ArrayDifferenzaB.diff(new int[]{1, 3, 5, 7, 9},
    new int[]{1, 5, 7}));
}
```

Test Case 2: The first array is empty, and the second array contains some elements.

```
@Test
public void testEmptyArray() {
    assertArrayEquals(new int[]{},
    ArrayDifferenza3.diff(new int[]{},
    new int[]{1, 2, 3}));
}
```

Test Case 3: Both input arrays are empty.

```
@Test
public void testBothEmptyArrays() {
    assertArrayEquals(new int[]{},
    ArrayDifferenza4.diff(new int[]{},
    new int[]{}));
}
```

Test Case 4: All the elements in the first array are also present in the second array.

```
@Test
public void testAllCommonElements() {
    assertArrayEquals(new int[]{},
    ArrayDifferenza3.diff(new int[]{1, 2, 3},
    new int[]{1, 2, 3}));
}
```

Test Case 5: There are no common elements between the two arrays.

```
@Test
public void testNoCommonElements() {
    assertArrayEquals(new int[]{1, 2, 3},
    ArrayDifferenza3.diff(new int[]{1, 2, 3},
    new int[]{4, 5, 6}));
}
```

Test Case 6: There are duplicate elements in the first array, identifying and returning only one instance of each unique element that is not present in the second array.

```
@Test
public void testDuplicates() {
    assertArrayEquals(new int[]{1},
    ArrayDifferenza4.diff(new int[]{1, 1, 2, 2},
    new int[]{2, 3, 4}));
}
```

Test Case 7: The arrays contain negative numbers, returning elements that are unique to the first array.

```
@Test
public void testNegativeNumbers() {
    assertArrayEquals(new int[]{-1},
    ArrayDifferenza4.diff(new int[]{-1, -2, -3},
    new int[]{-2, -3, -4}));
}
```

Test Case 8: The first array is null; it expects an exception.

```
@Test(expected = IllegalArgumentException.class)
public void testNullArray() {
    ArrayDifferenzaB.diff(null, new int[]{1, 5, 7});
}
```

Test Case 9: Both input arrays are null; it expects an exception.

```
@Test(expected = IllegalArgumentException.class)
public void testBothNullArrays() {
    ArrayDifferenzaB.diff(null, null);
}
```

Problem 2 “ComparaSequenze" Test Cases:
Test Case 1: Initializes and returns a non-null array, even if the list might be empty; it does not return null.

```
@Test
public void testAcquisisciSequenzaRealiNotNull() {
    assertNotNull("Sequenza A should not be null",
    ComparaSequenze3.acquisisciSequenzaReali());
}
```

Test Case 2: Returns an array containing a specific predefined set of real numbers; it validates both the presence and the order of these numbers.

```
@Test
public void testAcquisisciSequenzaRealiContents() {
    ArrayList<Double> expected = new ArrayList<>();
    expected.add(1.0);
    expected.add(2.0);
    expected.add(3.0);
    expected.add(4.0);
    assertEquals(expected,
    ComparaSequenze3.acquisisciSequenzaReali());
}
```

Test Case 3: Returns a non-null array of type Frazione3; it initializes and returns a list.

```
@Test
public void testAcquisisciSequenzaFrazioniNotNull() {
    assertNotNull("Sequenza B should not be null",
    ComparaSequenze3.acquisisciSequenzaFrazioni());
}
```

Test Case 4: Returns an array containing specific predefined fractions; it checks for the presence and the instantiating of these fraction objects (Frazione3) within the list.

```
@Test
public void testAcquisisciSequenzaFrazioniContents() {
    ArrayList<Frazione3> expected = new ArrayList<>();
    expected.add(new Frazione3(1, 10));
    expected.add(new Frazione3(100, 2));
    expected.add(new Frazione3(100, 100));
    expected.add(new Frazione3(5, 4));
    assertEquals(expected,
    ComparaSequenze3.acquisisciSequenzaFrazioni());
}
```

Test Case 5: Calculates the half of the sum of the provided elements in the list and compares it to an expected value to ensure accuracy, with a non-empty list of real numbers.

```
@Test
public void testCalcolaMetaMinore() {
    ArrayList<Double> sequenzaA = new ArrayList<>();
    sequenzaA.add(2.0);
    sequenzaA.add(4.0);
    sequenzaA.add(6.0);
    double expected = 6.0;
    assertEquals(expected,
    ComparaSequenze3.calcolaMetaMinore(sequenzaA), 0.001);
}
```

Test Case 6: Tests an empty list to ensure it handles such cases and returns 0.0, indicating no elements.

```
@Test
public void testCalcolaMetaMinoreEmptyList() {
    ArrayList<Double> sequenzaA = new ArrayList<>();
    double expected = 0.0;
    assertEquals(expected,
    ComparaSequenze3.calcolaMetaMinore(sequenzaA), 0.001);
}
```

Test Case 7: Verifies the functioning of the Frazione3 constructor; it ensures that a non-null instance of Frazione3 and that the fraction structure (numerator and denominator) is set according to the provided arguments.

```
@Test
public void testFrazione3Constructor() {
    Frazione3 frazione = new Frazione3(2, 3);
    assertNotNull("Frazione3 object should not be null",
    frazione);
    assertEquals(2, frazione.getNumeratore());
    assertEquals(3, frazione.getDenominatore());
}
```

Test Case 8: Compares two Frazione3 objects and returns true if the first fraction is greater than the second.

```
@Test
public void testFrazione3IsMaggiore() {
    Frazione3 frazione1 = new Frazione3(1, 2);
    Frazione3 frazione2 = new Frazione3(1, 3);
    assertTrue("1/2 should be greater than 1/3",
    frazione1.isMaggiore(frazione2));
}
```

"MatriceStringa" Problem 3 Test Cases:
Test Case 1: Verifies that a MatriceStringa3 object can be created with valid dimensions and a default value, ensuring that the resulting matrix object is not null.

```
@Test
public void testValidMatrixCreation() {
    MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
    assertNotNull("Matrix should not be null", matrice);
}
```

Test Case 2: Verifies the matrix creation with zero rows; it expects an exception.

```
@Test(expected = RuntimeException.class)
public void testMatrixCreationWithZeroRows() {
    new MatriceStringa3(0, 3, "test");
}
```

Test Case 3: Verifies the matrix creation with zero columns; it expects an exception.

```
@Test(expected = RuntimeException.class)
public void testMatrixCreationWithZeroColumns() {
    new MatriceStringa3(3, 0, "test");
}
```

Test Case 4: Verifies the matrix creation with a negative number of rows; it expects an exception.

```
@Test(expected = RuntimeException.class)
public void testMatrixCreationWithNegativeRows() {
    new MatriceStringa3(-1, 3, "test");
}
```

Test Case 5: Verifies the matrix creation with a negative number of columns; it expects an exception.

```
@Test(expected = RuntimeException.class)
public void testMatrixCreationWithNegativeColumns() {
    new MatriceStringa3(3, -1, "test");
}
```

Test Case 6: Verifies setting the value of a cell within the matrix bounds.

```
@Test
public void testSetValidCell() {
    MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
    matrice.set(1, 1, "newValue");
}
```

Test Case 7: Verifies setting a cell with a row index out of bounds; it expects an exception.

```
@Test(expected = MatriceStringa3.MatriceException.class)
public void testSetCellWithRowOutOfBounds() {
    MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
    matrice.set(3, 1, "value");
}
```

Test Case 8: Verifies setting a cell with a negative row index; it expects an exception.

```
    @Test(expected = MatriceStringa3.MatriceException.class)
    public void testSetCellWithNegativeRow() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        matrice.set(-1, 1, "value");
}
```

Test Case 9: Verifies setting a cell with an out of bounds column index; it expects an exception.

```
    @Test(expected = MatriceStringa3.MatriceException.class)
    public void testSetCellWithColumnOutOfBounds() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        matrice.set(1, 3, "value");
}
```

Test Case 10: Verifies setting a cell with a negative column index; it expects an exception.

```
    @Test(expected = MatriceStringa3.MatriceException.class)
    public void testSetCellWithNegativeColumn() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        matrice.set(1, -1, "value");
}
```

Test Case 11: Converts a matrix row to a string with a valid separator.

```
    @Test
    public void testRigaToStringValid() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        String expected = "test.test.test";
        assertEquals(expected, matrice.rigaToString(1, "."));
    }
```

Test Case 12: Converting a row to a string with an out-of-bounds index; it expects an exception.

```
    @Test(expected = MatriceStringa3.MatriceException.class)
    public void testRigaToStringWithHighIndex() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        matrice.rigaToString(3, ".");
    }
```

Test Case 13: Converting a row to a string with a negative index; it expects an exception.

```
    @Test(expected = MatriceStringa3.MatriceException.class)
    public void testRigaToStringWithNegativeIndex() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        matrice.rigaToString(-1, ".");
```

Test Case 14: Separatore is null; it expects an exception.

```
    @Test(expected = MatriceStringa3.MatriceException.class)
    public void testRigaToStringWithNullSeparator() {
        MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
        matrice.rigaToString(1, null);
    }
```

Test Case 15: Separatore is an empty string; it expects an exception.

```
@Test
public void testRigaToStringWithEmptySeparator() {
    MatriceStringa3 matrice = new MatriceStringa3(3, 3, "test");
    String expected = "testtesttest";
    assertEquals(expected, matrice.rigaToString(1, ""));
}
```

Test Case 16: Verifies that the matrix creates null values, and that the cells are set to null and retrievable.

```
@Test
public void testMatrixCreationWithNullValue() {
    MatriceStringa3 matrice = new MatriceStringa3(3, 3, null);
    assertNull("Matrix cell should be initialized with null",
    matrice.get(0, 0));
}
```

Test Case 17: Verifies the matrix creation with empty strings as the initial value for cells, and that the rows convert to a string representation, even when the cells are empty.

```
@Test
public void testMatrixCreationWithEmptyString() {
    MatriceStringa3 matrice = new MatriceStringa3(3, 3, "");
    assertEquals("", matrice.rigaToString(0, ","));
}
```

References

1. Beganovic, A.; Jaber, M.A.; Abd Almisreb, A. Methods and Applications of ChatGPT in Software Development: A Literature Review. *Southeast Eur. J. Soft Comput.* **2023**, *12*, 8–12. https://doi.org/10.21533/scjournal.v12i1.251.
2. Jamdade, M.; Liu, Y. A Pilot Study on Secure Code Generation with ChatGPT for Web Applications. In Proceedings of the 2024 ACM Southeast Conference, ACM SE'24, Marietta, GA, USA, 18–20 April 2024; pp. 229–234. https://doi.org/10.1145/3603287.3651194.
3. Guo, Q.; Cao, J.; Xie, X.; Liu, S.; Li, X.; Chen, B.; Peng, X. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE'24, Lisbon Portugal, 14–20 April 2024. https://doi.org/10.1145/3597503.3623306.
4. Jeuring, J.; Groot, R.; Keuning, H. What Skills Do You Need When Developing Software Using ChatGPT? (Discussion Paper). In Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, Koli Calling'23, Koli, Finland, 13–18 November 2023. https://doi.org/10.1145/3631802.3631807.
5. Liu, J.; Xia, C.S.; Wang, Y.; Zhang, L. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *arxiv* **2023**, arXiv:cs.SE/2305.01210.
6. EvalPlus Team. EvalPlus. 2023. Available online: https://evalplus.github.io (accessed on 18 May 2024).
7. Khoury, R.; Avila, A.R.; Brunelle, J.; Camara, B.M. How Secure is Code Generated by ChatGPT? *arXiv* **2023**, arXiv:cs.CR/2304.09655.
8. OpenAI. ChatGPT: Optimizing Language Models for Dialogue. 2022. Available online: https://openai.com/chatgpt (accessed on 18 May 2024).
9. Liu, Z.; Tang, Y.; Luo, X.; Zhou, Y.; Zhang, L. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Trans. Softw. Eng.* 2024, *early access*. https://doi.org/10.1109/TSE.2024.3392499.
10. GitHub. CodeQL. 2005. Available online: https://codeql.github.com/ (accessed on 18 May 2024).
11. LeetCode. 2015. Available online: https://leetcode.com/ (accessed on 18 May 2024).
12. Liu, Y.; Le-Cong, T.; Widyasari, R.; Tantithamthavorn, C.; Li, L.; Le, X.B.D.; Lo, D. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Trans. Softw. Eng. Methodol.* 2024, *accepted*. https://doi.org/10.1145/3643674.
13. GitHub. GitHub Copilot. 2021. Available online: https://github.com/features/copilot (accessed on 18 May 2024).
14. Amazon. Amazon CodeWhisperer. 2023. Available online: https://aws.amazon.com/codewhisperer/ (accessed on 18 May 2024).

15. Yetiştiren, B.; Özsoy, I.; Ayerdem, M.; Tüzün, E. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv* **2023**, arXiv:cs.SE/2304.10778.
16. Chen, E.; Huang, R.; Chen, H.; Tseng, Y.H.; Li, L.Y. GPTutor: A ChatGPT-powered programming tool for code explanation. In Proceedings of the International Conference on Artificial Intelligence in Education, Tokyo, Japan, 3–7 July 2023.
17. Tian, H.; Lu, W.; Li, T.O.; Tang, X.; Cheung, S.C.; Klein, J.; Bissyandé, T.F. Is ChatGPT the Ultimate Programming Assistant—How far is it? *arXiv* **2023**, arXiv:cs.SE/2304.11938.
18. Sakib, F.A.; Khan, S.H.; Karim, A.H.M.R. Extending the Frontier of ChatGPT: Code Generation and Debugging. *arXiv* **2023**, arXiv:cs.SE/2307.08260.
19. Feng, Y.; Vanam, S.; Cherukupally, M.; Zheng, W.; Qiu, M.; Chen, H. Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. In Proceedings of the 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), Torino, Italy, 26–30 June 2023; pp. 876–885. https://doi.org/10.1109/COMPSAC57700.2023.00117.
20. Cordasco, I.S. Flake8. 2010. Available online: https://flake8.pycqa.org/en/latest/ (accessed on 18 May 2024).
21. Google. Bard. 2023. Available online: https://bard.google.com/chat (accessed on 18 May 2024).
22. Microsoft. Bing Copilot AI. 2023. Available online: https://www.microsoft.com/en-us/bing?ep=0&form=MA13LV&es=31 (accessed on 18 May 2024).
23. Meta. Llama. 2023. Available online: https://llama.meta.com/ (accessed on 18 May 2024).
24. Anthropic. Claude. 2023. Available online: https://claude.ai/ (accessed on 18 May 2024).
25. SonarSource. SonarQube. 2006. Available online: https://www.sonarsource.com/products/sonarqube/ (accessed on 18 May 2024).
26. SonarSource. SonarCloud. 2006. Available online: https://www.sonarsource.com/products/sonarcloud/ (accessed on 18 May 2024).