IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 8 April 2024, accepted 14 May 2024, date of publication 21 May 2024, date of current version 30 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3403858

RESEARCH ARTICLE

Methodology for Code Synthesis Evaluation of LLMs Presented by a Case Study of ChatGPT and Copilot

ZOLTÁN SÁGODI[1,2], ISTVÁN SIKET[1,2], AND RUDOLF FERENC[1]
[1]Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary
[2]FrontEndART Software Ltd., 6721 Szeged, Hungary

Corresponding author: Zoltán Ságodi (sagodiz@inf.u-szeged.hu)

This work was supported in part by European Union Project within the Framework of the Artificial Intelligence National Laboratory under Grant RRF-2.3.1-21-2022-00004; in part by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA Funding Scheme under Project TKP2021-NVA-09; in part by European Union under the Horizon Europe Programme (AI4CYBER) under Grant 101070450; and in part by the University of Szeged Open Access Fund under Grant 6537.

ABSTRACT Large Language Models (LLMs) have grown in popularity in recent years and are now employed in a variety of software engineering domains thanks to their Natural Language Processing (NLP) capabilities, which include source code generation, understanding, and documentation. Selecting the appropriate model for source code generation presents a problem to developers as more and more powerful LLMs become available. While some studies have evaluated Copilot or ChatGPT, there is a lack of research on how developers can choose from available LLMs, which is a key factor in the growing set of available models and services. It is crucial to know if a model is capable of generating useful source code that meets the quality requirements and if the developers will be able to use the generated code. Regarding these factors, one has to decide which model to utilize during everyday tasks. This paper shows a methodology to compare such models by demonstrating an actual comparison of two models. Subsequently, we investigated the functional and non-functional qualities of the code synthesized by the models on a program synthesis benchmark containing 25 tasks. On average, the functional testing shows that ChatGPT generated 17 perfect solutions, while Copilot could only solve 13. The non-functional analysis reflected that both models generated good quality code, however, both have characteristic code smells. Our evaluation shows that ChatGPT performs better using this methodology, which is supported by human reviewers who evaluated the generated code by hand.

INDEX TERMS Artificial intelligence, copilot, ChatGPT, code-synthesis, code quality, large language models, model selection.

I. INTRODUCTION

In everyday life, Large Language Models (LLMs) are becoming increasingly popular and are used for various purposes [1]. It is the same in software engineering as developers leverage LLMs' Natural Language Processing (NLP) capabilities. LLMs are used for a variety of tasks such as source code comprehension, documentation generation, or various testing tasks, e.g. test case generation. In recent years, source code generation has emerged as a key component of LLM utilization.

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

Source code generation is the process of automatically producing code based on high-level abstractions defined in domain-specific languages (DSLs) [2] or other declarative languages. The objective of source code generation is to reduce the amount of low-level, repetitive coding work that developers need to do, enabling them to focus on higher-level tasks and reducing the likelihood of errors and bugs. While source code generation has a long history, from the 70's [3] even to the 2010s [4], recent advances in LLMs [5] allow developers to express required program code using natural language definitions.

While having more time for abstract tasks and generating code with low cost is advantageous, in software development, the quality of the source code is at least as important as the time spent on development. There are multiple factors affecting source code quality, and there are several ways to measure it. One approach is using static code analysis [6], which allows for the code to be analyzed without executing it, without any tests, or even without a compiled binary file. The results of the analysis can be coding rule violations, detected code duplication, or code metrics. Although static analysis is a powerful and widely used technique, it is not the only way to measure source code quality. Code quality is often defined from a human perspective, and metrics may not adequately reflect this [7].

The previously mentioned aspects are the keys to successfully using LLMs in software engineering, although it is not shown how to choose the best LLM for a given task. There are already studies that evaluated a chosen LLM [5], [8] from various perspectives, but evaluating only one LLM might mean missing out on a better model. Evaluating all the possible models by the previously described methods takes too much effort as they mostly require a lot of human actors and effort during the evaluation.

In this paper, we investigate different aspects of the quality of LLM generated source code. It is important to know how to compare the generated source code; what are the main factors in which they differ most; and what aspects a developer should consider when choosing a development assist tool.

We do this by giving a simple methodology and showing its usefulness through an actual evaluation of two LLMs capable of source code generation: ChatGPT and Copilot. We examine how these models work with natural language task definitions, how the generated code (for C++ and Java) behaves, and what the quality of the generated code is like from multiple perspectives. We decide which model is better in every phase of the evaluation and finally we support our decisions by human reviewing, similar to other papers [8], [9]. The research questions addressed in this paper are:

- **RQ 1: How does LLM-generated source code score in terms of source code quality?**
- **RQ 2: Is the generated source code accepted by developers?**
- **RQ 3: What aspects should be considered when choosing LLM-based generative tools?**

The main goal of this paper is to highlight the importance of proper comparisons, not to introduce optimized prompting or selecting up-to-date models as those are changing dynamically. The latter might be decided using the provided methodology with actualized data and models.

This topic relates to RQ3, in which we discuss the main components of a decent comparison, but it cannot be discussed without the previous two RQs. Our first RQ is in the first place because humans will eventually operate with source code, where quality still matters as people prefer to work with good quality code. In our use case, we found that there are models that are capable of generating code with good quality. Our second RQ is still required in our methodology, as various problems can be solved in various ways and programming manners. Developers usually have their favorable style and way of thinking. The use case showed that developers tend to accept the generated code conditionally, meaning that the generated code is not perfect but can be used by developers and it helps their work.

The paper's subsequent sections describe related works in Section II. We present our methodology in Section III, and in Section IV, we compare two models in an actual case study and discuss the results. Section V addresses threats to validity. Finally, we summarize the paper in Section VI.

II. RELATED WORK

In this work, we compare LLMs based on the quality of their generated source code from different aspects. Both LLMs and quality assurance are already integral parts of software engineering both in academic [10], [11] and industrial fields [12].

There are various papers describing static analysis techniques and tools [13], [14], articles that compare these tools [15], and even papers that use machine learning for static analysis tasks [16]. Static analysis is performed on the source code (or the generated byte code in the case of Java-based languages) thus it does not require running the code. The code is mostly written by developers, but nowadays source code generation and synthesis allow developers to create code without any or with only a minimal amount of human interaction. As source code synthesis has a long history [17], those methods used to rely on mathematical derivations and various rules. Nowadays, neural networks are taking over these methods since LLMs are available such as Codex [5], which supports GitHub's Copilot, and the recently introduced ChatGPT. Both models are based on GPT-3 [18]. There are also models based on Google's LaMDA [19], such as Bard. These models are generative models [20], meaning they are capable of producing text from an input, usually natural language text, and the output can be source code.

As quality assurance and static analysis can be performed on the source code it is obvious that we must examine how the generated code performs. There are works that evaluate LLMs. Vaithilingam et al. [8] evaluate Codex via the Copilot plugin and show the effects of such a tool on the development itself. They included 24 developers and asked them to create a program using Intellisense and Copilot. Sobania et al. [21] did a comparative study on Copilot and Genetic program synthesis. They used the same benchmark [22] we did; however, they added extra prompting by defining a function's signature, thereby optimizing the prompt for Copilot. Their main goal was to compare two methodologies for program synthesis and did not measure the quality of the synthetized code. In the work of Madi [9], the readability is measured by

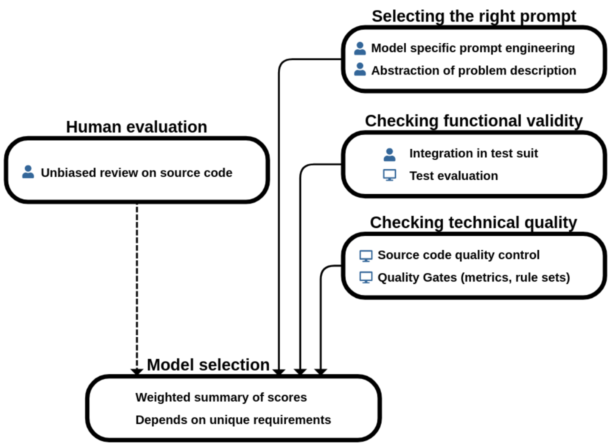
Selecting the right prompt
Model specific prompt engineering
Abstraction of problem description
Human evaluation
Unbiased review on source code
Checking functional validity
Integration in test suit
Test evaluation
Checking technical quality
Source code quality control
Quality Gates (metrics, rule sets)
Model selection
Weighted summary of scores
Depends on unique requirements

FIGURE 1. Overview of the methodology with key elements.

static code analysis but they do not check further qualities and they rely heavily on human annotators; 21 people to be specific. These works show that LLMs are capable of generating source code that is similar to human-written code in terms of readability and maintainability. There are even works [23] showing the performance of the generated code matches human-written code. Although there are works that evaluate not only one LLM [24], that highlight the drawbacks of LLMs, they do not show a proper way to compare them, therefore, they give no suggestion for how the better model could be chosen. These works mostly rely on a larger number of annotators or human reviewers, which is an expensive resource both in the academic and especially in industrial field.

LLMs are also used to fix vulnerabilities in source code [25], which might suggest that these models are free from vulnerabilities, however, there are multiple works that show Copilot being prone to generate vulnerable code [26], [27], [28]. ChatGPT is not free from vulnerabilities either as it is shown in the work of Khoury et al. [29]. In their work, it is shown that ChatGPT is capable of fixing the vulnerabilities by further prompting, which leads us to the problem of optimal prompting.

III. METHODOLOGY

In this section, we describe a methodology of how LLMs can be compared and which are the main factors to consider. We describe four phases: selecting the right prompt, checking functional validity, checking technical quality, and human evaluation. The main goal of this paper is to emphasize the importance of each step. The main steps and key elements are shown in Figure 1.

A. SELECTING THE RIGHT PROMPT

First of all, the prompt provided for the models must be carefully selected, which presents a challenge as none of the compared models should be favored by optimized prompting. The difficulty and level of detail of the problems' descriptions are the main factors to consider when working with NLP-based systems.

The difficulty of the problem greatly affects the quality and usability of the generated source code. A simple "hello world" program provides no meaningful information about the models' capabilities. When attempting to generate a complete project with multiple files and perform complex algorithms, the model may encounter difficulties that prevent it from generating proper source code. This way the model's capabilities stay hidden. The input task must have the appropriate level of difficulty to measure the difference between models.

The level of detail in the problem description is also a crucial factor. Providing too much information gives no indication of whether the model can identify complex connections between elements within the problem. Providing too little information may result in the models not generating a solution, or the solutions may be too different, which does not provide any additional information for our comparison. The detail of the description must be at a level that challenges the models, but it should be clear from a description what the solution should be.

Another crucial factor in selecting the right prompt is prompt engineering. Although LLMs are designed to interact with natural language, therefore, simple commands, such as "Generate", or "Write code" can lead to the desired results, various prompting techniques could optimize these results. Such techniques must be taken into consideration, paying attention to the latest prompt engineering results relating to the models; e.g., Github's advice [30]. Denny et al. [31] show that NL prompt changes affect the resulting source code. It can be performed with prompt patterns such as the work of White et al. [32].

B. CHECKING FUNCTIONAL VALIDITY

Secondly, after using the right prompts, the LLM-generated source code has to be syntactically correct and fully functional. To ensure this, functional testing has to be performed which must include various test cases both for selected edge cases and non-edge cases as this shows how robust the solution is. To perform this testing, proper test cases need to be available with input and required output pairs. Handling the input and output may be different in the code generated by different LLMs, so one might have to modify the generated code for automated tests, and the modifications must not have any effect on the functional code parts.

C. CHECKING TECHNICAL QUALITY

Thirdly, the generated code has to pass some quality requirements, which can be measured by static analysis. The static analysis could include various techniques, rules, and metrics. All of these elements depend on what context the generated code is used in. An optimal solution would use the same pipeline for static analysis as the actual development. If no such configuration is available, static analyzers should be configured according to best practices, e.g. by considering the work of Kaner and Bond [7].

D. HUMAN EVALUATION

Finally, the most expensive part is human evaluation. This phase is optional as human resources are expensive, however, one can be fully confident in the results after human reviews. In contrast to other works, we do not suggest that a large set of developers should evaluate the generated source code as this phase is just additional to the previous phases. During human evaluation, reviewers must not know about the result of the functional test phase and technical test phase. Reviewers should be asked to evaluate the generated source code on an even-numbered scale, forcing them into decision making.

Although the methodology steps seem trivial, many of the related works lack this evaluation methodology. Vaithilingam et al. [8] and Dominik et al. [21] did evaluate program synthesis, but the results were conducted on functional results and the experience of the users, while there were no source code quality, security, or performance comparisons. Although, in the work of Madi [9], they evaluate the static properties of the source code, they lack the proper description of functional properties and the way of prompting during the iterative development. Related works lack the proper evaluation steps on the very basic steps, therefore, an abstract description must be given in order to properly compare source code synthetization methods.

IV. CASE STUDY

In this section, we compare two LLMs, namely Copilot and ChatGPT using the methodology presented in Section III. This case study is a simplified version of an actual comparison in order to show how to evaluate actual models. An actual evaluation must be done on a benchmark that fits the required programming qualities the best, such as graphical pipeline programming, high-end servers, low-latency requirements, etc. During an actual evaluation, it is not the default SonarQube settings that should be used, but rather the available unique rule sets, configurations or even other, already-in-use static analyzers apart from SonarQube.

Selecting Copilot and ChatGPT was not based solely on their popularity in software engineering tasks but on their applicability. They have great availability through APIs and UIs and, therefore, are likely to be used in actual projects. Although there are other source code specific LLMs, such as CodeT5- or BERT-based models, which can be evaluated with our methodology, due to their limited availability through APIs they are less likely to be integrated. We provide in the supplementary material the generated source code and the applied diffs to them. We provide the scripts and random numbers we used during our research to run the benchmark evaluations. We also include the validated results for static analysis and the raw numbers from the human evaluation phase. To this case study we provide an online appendix available on Zenodo: https://zenodo.org/record/8123647.

A. SETUP OF THE CASE STUDY

To present the usage of our methodology we had to consider various factors such as which models to compare and what programming language to evaluate these models on. In the following sections, we describe which models were selected and why. We also describe our programming language selection.

1) UTILIZED LLMS

For our comparison, we utilized Copilot, a Visual Studio Code extension[1] and ChatGPT,[2] which we accessed through its online web interface. Both models were employed to generate input data on the first week of January 2023. As these models can interact with developers, there is no definitive answer to any given problem. Copilot continuously generates code snippets as the developer writes code, while ChatGPT can modify the code if the developer provides additional conditions or a more precise description of the task. To ensure a fair comparison, we used the same problem description for both models and did not interact further with them. This means that we accepted the first generated code for a given input. Using Copilot we ensured that the workspace is clean so the results were not altered. Using ChatGPT we opened a new chat windows for every task in order to have a clean context.

2) PROGRAMMING LANGUAGE SELECTION

In terms of programming languages, we had to choose from multiple languages due to limited manpower during the comparison evaluations. We selected languages that are widely used in academic and industrial fields. Our first language of choice was C++, as it is easy to program using classes and an object-oriented programming (OOP) style or using only functions in a non-OOP style. Additionally, C++ provides a wide range of tools that are easy to use but can also be used in a complex manner. Furthermore, C++ is notorious for its memory management issues, making it a good choice for our code quality tests.

To mitigate the potential effects of language selection on our results, we decided to use another language that is a managed language where memory management issues do not arise and the language primarily supports OOP. Our second language of choice was Java, which is a widely used programming language, and it is well-known for having a lot of boilerplate code. Asking for Java code requires the model to generate every small detail to have a working solution, making Java a good choice for our second language.

To note, C++ and Java are both compiled languages, which is an additional condition against the generated sources as the code must compile.

[1]https://marketplace.visualstudio.com/items?itemName=GitHub.copilot
[2]Version Dec 15 - https://chat.openai.com/.

```
Generate a C++ code to solve the problem!
Given a vector of integers, return the
first index such that the sum of all
integers from the start of the vector to
that index (inclusive) is negative
```

LISTING 1. Example prompt for a C++ solution for the basement problem.

B. SELECTING THE RIGHT PROMPT

The first step in our methodology is selecting the right prompts with right difficulty and level of detail. As this task is very complicated we decided to use a program synthesis benchmark [22], which includes 25 programming tasks from various sources. We provided the task descriptions from this benchmark as the prompts with a fixed prefix e.g. "Generate a <LANGUAGE> code to solve the problem!", where LANGUAGE is C++ or Java. A C++ prompt example is shown in Listing 1, where the first sentence is the prefix with the selected language and the remaining part is the text extracted from the benchmark.

We decided to use simple prompts as the case study is just a way of showing our methodology in action. The selected prompts are likely to be used by a software developer who does not have prompt engineering experience. The person evaluating will not necessarily use these exact prompts but instead use specific techniques that may not even exist yet. They might be using a model that needs a special kind of prompting (e.g. they have to provide elements in JSON because that is how the model was trained). Therefore, since we cannot provide for all current and future use cases, optimizing the prompts was not something we were aiming for.

C. CHECKING FUNCTIONAL VALIDITY

To evaluate the program's functional validity, we used the benchmark's input and output values for each task. These values were divided into two disjunct parts: edge cases, which included various tests for each task that were considered edge cases for the problem, and random cases. The number of edge cases varied from 5 to 40 depending on the task. We executed all available edge case tests. The random test cases included one million input-output pairs per task. We randomly[3] selected 10,000 input-output pairs from the pool of one million random cases for every task. The selected input-output pairs were used as random cases for later evaluations.

However, the generated source code was not directly usable for executing the tests because it might or might not read inputs from the console or write results to the console. Therefore, we had to further modify the code to unify the inputs and outputs by using the same way of reading input from the console and writing output to the console. It also included that additional text answers were removed and only the required parts were kept. Only the input and output lines were modified, not the functional code. All source code differences are available.[4]

Java source code did not require additional modifications. In the case of C++, ChatGPT generated solutions mostly compiled with no error, but we had to subsequently correct the generated programs for 3 benchmark tasks manually:

- *leaders*: include `climits`, `algorithm` were missing.
- *cut_vector*: `climits` include was missing.
- *vector_distance*: `limits` include was missing.

All these errors are related to missing includes, which could be solved by a simple added line. There were no problems during the linking phase after the missing includes were added.

Copilot also generated mostly compilable C++ code, but there were also some problems that we had to handle:

- *leaders*: It generated a vector variable with the same name as a previously declared function, thus resulting in a compile error as the function call ended up being a call-operator of the declared variable. Renaming the variable solved the error.
- *indices_of_substring*: The generated source did not contain a `main` method, but only a function containing the solution, thus resulting in a linker fault. This problem can be solved by using a dummy `main` method.
- *twitter*: The generated source did not contain a `main` method. We used the same solution as for *indices_of_substring*.

Once the programs were ready to run and the input and output values were unified, we were able to compare the desired and actual output values. We performed a strict comparison, meaning we compared each character in the output values. The benchmark also included floating-point values with large precision as outputs, so numeric errors could lead to different values even with correct functional code. To address this issue we compared floating-point values with an epsilon value, and differences smaller than the given epsilon did not count as differing values. The epsilon value for comparison was 0.00001, which was determined by investigating the differences between programmed computations and closed formula computations for the tasks when it was possible. We use these results as epsilon evaluation results.

Tables 1 and 2 present the performance results for C++ and Java, displaying the pass ratio for each model on every task. The tables include edge case, random case tests and provide results for exact matching, as well as epsilon matching, which was included to accommodate comparisons involving floating point numbers. Furthermore, the "Diff" columns in the tables provide the difference in pass ratio between the two models on the exact and epsilon matching. This difference value indicates whether Copilot or ChatGPT performed better. Specifically, a negative difference value indicates better performance from Copilot, while a positive value suggests better performance from ChatGPT. The cells

[3] See supplementary material referenced in Section IV - https://zenodo.org/record/8123647.

[4] See supplementary material referenced in Section IV.

are colored, and the bluer a cell is, the better Copilot performed; the more orange a cell is, the better ChatGPT performed.

Later on, we use the epsilon results as they are not altered by the floating point comparison error. Although, the results are very similar in exact and random comparisons. There are a few cases where epsilon comparison resulted in a larger difference:

- C++ - bouncing_balls
- C++ - snow_day
- Java - bouncing_balls
- Java - vector_distance
- Java - snow_day

The tasks bouncing_balls and snow_day are tasks where the generated source code contains iterative computation with floating-point variables, therefore, the floating-point error is not a surprise. The task vector_distance includes large dimensional vectors with floating point values, therefore, the floating-point error could cause differences in the outputs.

1) C++

During the edge case testing phase (see Table 1), a number of solutions encountered runtime errors. Specifically, the ChatGPT model faced runtime errors in 80% of the test cases for the solve_boolean task. After conducting a thorough investigation, we determined that an incorrect operator ordering was the root cause of the issue, which subsequently impacted the entire testing process whenever an operator was encountered. On the other hand, the Copilot model only experienced a single instance of runtime error during edge case testing, which occurred with no input.

Despite these challenges, ChatGPT achieved pass rates of 100% with epsilon evaluation on 18 out of 25 tasks, while Copilot achieved pass rates of 100% with epsilon evaluation on 13 out of 25 tasks. Notably, the models obtained identical pass-rates on 14 out of 25 tasks. However, on average, Copilot outperformed ChatGPT with a pass-rate of 54% on three tasks, with a standard deviation of 36.14, while ChatGPT performed better with an average pass-rate of 58% on eight tasks, with a standard deviation of 35.98.

Although edge cases are an integral part of software testing, it is important to investigate how the generated sources perform with a larger input size. On the random test set (see Table 1), ChatGPT managed to achieve a 100% pass ratio on 19 tasks, whereas Copilot achieved a 100% pass ratio on 13 tasks.

Examining the epsilon differences, ChatGPT exhibited superior performance with an average pass-rate of 72.88% on eight tasks, with a standard deviation of 35.92. Likewise, Copilot performed better on two tasks, with an average pass-rate of 51% and a standard deviation of 69.3. Based on the deviation, it is evident that ChatGPT outperforms Copilot in a more consistent manner.

Upon closer examination of the data, it appears that performing random tests instead of edge cases did not unconditionally increase the pass ratio. In fact, it resulted in a decrease in a few tasks. Solutions generated by ChatGPT with lower pass rate on random test cases were: *bouncing_balls, bowling, dice_game, solve_boolean, vector_distance*.

Only on one of these tasks did the ChatGPT-generated source code reach 100% during edge case testing. Since tasks with a pass-rate of less than 100% could potentially worsen by increasing the number of erroneous cases, the task that originally reached the pass rate of 100% (*vector_distance*) was of particular interest. Upon further investigation, we discovered that the discrepancies in this solution were caused by floating-point errors exceeding our epsilon value.

Solutions generated by Copilot with lower pass rate on random test cases were: *bouncing_balls, bowling, dice_game, find_pair, leaders, snow_day, vector_distance*.

In the case of Copilot, *vector_distance* was the only task that reached 100% pass rate previously and lowered in the random case. This disparity can be attributed to discrepancies in floating-point values that exceeded our epsilon value. Notably, the values changed in unison for ChatGPT and Copilot relating to this task, and the generated solutions were nearly identical, save for the fact that Copilot lacked vector dimension validation, which was not an explicit requirement of the task.

We looked into how the models performed in both edge and random case testing, and also how the pass-rates decreased. However, another important factor to consider in functional testing is whether the results remain similar or get better with random testing. If the results are similar during random testing, it suggests that the edge cases do not negatively impact the pass-rate, and thus the solution is generally more robust.

We found that ChatGPT performed better on five tasks on the random case than on edge case. The average difference was 21.4% with a standard deviation of 8.44. Copilot performed better on seven tasks, with an average of 39% on the random case with a standard deviation of 21.56. Since using random test cases mainly affected code that did not achieve a 100% pass rate in the edge case scenario, and using random tests has increased the pass rate especially for Copilot, we conclude that ChatGPT is more robust.

Regarding C++, ChatGPT performs slightly better on both edge case and random selected testing using epsilon-matching.

2) JAVA

We evaluated both models using Java language as well and the results are presented in Table 2, similarly to the C++ evaluation.

During edge case testing, the ChatGPT model encountered runtime errors in 5 out of 10 tests for the task *cut_vector*, and in 1 out of 10 tests for the task *leaders*. Meanwhile, the Copilot model encountered runtime errors in 13 out of 21 tests for the task *bowling*, and in 1 out of 10 tests for the task *leaders*.

TABLE 1. C++ - Edge case and random generated test pass ratio with exact and epsilon matching.

	Edge - Exact			Edge - Epsilon			Random - Exact			Random - Epsilon		
Task	ChatGPT	Copilot	Diff	ChatGPT	Copilot	Diff	ChatGPT	Copilot	Diff	ChatGPT	Copilot	Diff
basement	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
bouncing_balls	0%	**80%**	-80	10%	**80%**	-70	0%	**5%**	-5	**20%**	5%	15
bowling	**33%**	10%	23	**33%**	10%	23	0%	0%	0	0%	0%	0
camel_case	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
coin_sums	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
cut_vector	**100%**	0%	100	**100%**	0%	100	**100%**	0%	100	**100%**	0%	100
dice_game	20%	**33%**	-13	20%	**33%**	-13	1%	**3%**	-2	1%	**3%**	-2
find_pair	**100%**	84%	16	**100%**	84%	16	**100%**	28%	72	**100%**	28%	72
fizz_buzz	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
fuel_cost	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
gcd	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
indices_of_subs.	15%	15%	0	15%	15%	0	23%	23%	0	23%	23%	0
leaders	**100%**	50%	50	**100%**	50%	50	**100%**	18%	82	**100%**	18%	82
luhn	**100%**	0%	100	**100%**	0%	100	**100%**	0%	100	**100%**	0%	100
mastermind	**100%**	0%	100	**100%**	0%	100	**100%**	0%	100	**100%**	0%	100
middle_character	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
paired_digits	**100%**	65%	35	**100%**	65%	35	**100%**	81%	19	**100%**	81%	19
shopping_list	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
snow_day	50%	50%	0	**92%**	50%	42	0%	0%	0	**100%**	5%	95
solve_boolean	20%	**100%**	-80	20%	**100%**	-80	0%	**100%**	-100	0%	**100%**	-100
spin_words	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
square_digits	97%	97%	0	97%	97%	0	100%	100%	0	100%	100%	0
substitution_cip.	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
twitter	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
vector_distance	40%	40%	0	100%	100%	0	0%	0%	0	75%	75%	0

The runtime errors encountered by ChatGPT on the task *cut_vector* were due to improper handling of input where the code was not prepared for an array with only one element. On the other hand, the runtime error on task *leaders* was caused by the code's inability to handle an empty array input.

In the case of Copilot, the runtime errors encountered on the task *bowling* were caused by improper handling of string indices. The code was dereferencing characters based on an assumption that there would be more characters. For the task *leaders*, the code was not able to handle an empty array input.

Despite these errors, ChatGPT achieved a 100% pass rate with epsilon testing on 15 tasks, while Copilot achieved a 100% pass rate on 13 tasks during edge case testing. ChatGPT performed better on 10 tasks with an average pass-rate of 55% and a standard deviation of 31.77. In comparison, Copilot performed better on 4 tasks with an average pass-rate of 58% and a standard deviation of 40.4.

On random test cases, both ChatGPT and Copilot achieved a 100% pass rate on 14 tasks. During random testing ChatGPT performed worse on the following tasks than during edge case testing: *bouncing_balls bowling dice_game leaders snow_day solve_boolean vector_distance*. We conducted an investigation on the only task that attained a 100% pass rate on the edge case test, namely *vector_distance*. It was revealed that the discrepancies were due to floating point errors that exceeded the epsilon value employed in our matching method. Compared to edge case testing, Copilot performed worse on the *bowling, coin_sums, leaders, mastermind, shopping_list* and *snow_day* tasks. Among Copilot generated solutions there was no such task that reached 100% on the edge case testing and reached lower pass-rate on random testing.

Investigating the robustness of the generated code, using non-edge case tests compared to edge case tests, we found that ChatGPT's pass-rates increased on 2 tasks by an average of 22% with a standard deviation of 19.8, while Copilot's pass-rates increased by average of 9.67% with a standard deviation of 6.5. We can conclude that the source code generated by both models that reached 100% pass rate on the edge case mostly did not have lower pass-rates on random tests, and running non-edge test cases did not drastically increase the pass-rates. Therefore, the working solutions are robust. Regarding Java, ChatGPT performs slightly better, but Copilot's working examples seem to be more robust.

D. CHECKING TECHNICAL QUALITY

To perform a technical quality check, we employed static analyzers as a straightforward comparison method. We utilized SonarQube[5] and SonarScanner[6] to analyze the projects, and extended SonarQube with the SourceMeter plug-in [33] that offers additional metrics, coding rules, and language support. As the community edition of SonarQube does not support C++ analysis, we relied on SourceMeter for C++ analysis. This plug-in incorporates a variety of analyzers, such as ClangTidy, which are capable of detecting coding errors and bad programming practices of varying severity and calculating different metrics.

[5]V.8.9, www.sonarsource.com/products/sonarqube/downloads/lts/8-9-lts/.

[6]V.4.6.1.2450, binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-4.6.1.2450-linux.zip.

TABLE 2. Java - Edge case and random generated test pass ratio with exact and epsilon matching.

	Edge - Exact			Edge - Epsilon			Random - Exact			Random - Epsilon		
Task	ChatGPT	Copilot	Diff	ChatGPT	Copilot	Diff	ChatGPT	Copilot	Diff	ChatGPT	Copilot	Diff
basement	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
bouncing_balls	**80%**	0%	80	**80%**	10%	70	**5%**	0%	5	5%	**20%**	-15
bowling	**38%**	10%	28	**38%**	10%	28	**5%**	0%	5	**5%**	0%	5
camel_case	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
coin_sums	**100%**	9%	91	**100%**	9%	91	**100%**	0%	100	**100%**	0%	100
cut_vector	**50%**	0%	50	**50%**	0%	50	**86%**	0%	86	**86%**	0%	86
dice_game	73%	**100%**	-27	73%	**100%**	-27	35%	**65%**	-30	54%	**100%**	-46
find_pair	0%	**100%**	-100	0%	**100%**	-100	0%	**100%**	-100	0%	**100%**	-100
fizz_buzz	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
fuel_cost	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
gcd	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
indices_of_subs.	15%	**100%**	-85	15%	**100%**	-85	23%	**100%**	-77	23%	**100%**	-77
leaders	50%	50%	0	50%	50%	0	18%	18%	0	18%	18%	0
luhn	0%	0%	0	0%	0%	0	0%	0%	0	0%	0%	0
mastermind	**100%**	50%	50	**100%**	50%	50	**100%**	36%	64	**100%**	36%	64
middle_character	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
paired_digits	**100%**	65%	35	**100%**	65%	35	**100%**	81%	19	**100%**	81%	19
shopping_list	**100%**	10%	90	**100%**	10%	90	**100%**	0%	100	**100%**	0%	100
snow_day	**50%**	17%	33	**58%**	25%	33	0%	0%	0	**10%**	5%	5
solve_boolean	80%	**100%**	-20	80%	**100%**	-20	75%	**100%**	-25	75%	**100%**	-25
spin_words	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
square_digits	**100%**	97%	3	**100%**	97%	3	100%	100%	0	100%	100%	0
substitution_cip.	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
twitter	100%	100%	0	100%	100%	0	100%	100%	0	100%	100%	0
vector_distance	**60%**	0%	60	**100%**	0%	100	0%	0%	0	**98%**	0%	98

The coding smells and bugs were validated by hand as static analyzers are prone to false positive warnings and are tedious to configure. The material referenced in Section IV includes a list of reported warnings and the fact whether it was a true or false positive warning.

Although C++ and Java are object-oriented, the solutions were very simple and did not use anything from OOP, therefore, there was no use of measuring OOP metrics like inheritance, coupling, or cohesion, thus we selected size and complexity metrics only. We used several metrics, such as Logical Lines of Code (LLOC), Number of Statements (NOS), McCabe Cyclomatic Complexity (McCC), and Nesting Level (NLE), to evaluate the results of the static analysis. (LLOC excludes lines containing comments or whitespace. NLE counts multiple `else if` statements as only one additional depth.) These metrics are considered the lower the better.

On average, static analysis did not reveal any vulnerability hidden in the generated code either in ChatGPT- or in Copilot-generated solutions. The tasks were quite small and did not require a large amount of library usage, this way, the generated code appears to be vulnerability-free but cannot be considered so due to the non-deterministic nature of the models.

1) C++ RESULTS

The summarized results of the static analysis are presented in Table 3. For C++, it is evident that the models are comparable in terms of metrics. Therefore, we focused on a few cases where one model differs from the other more than the average. Additionally, aside from pure metrics, it is crucial to consider the code smells that these models generate since they may be prioritized.

The average ratio of the metrics is around 1.0 so we discuss the tasks where one model scored 1.5 times more of a metric than the other. These tasks are *cut_vector*, *dice_game*, *find_pair*, *fizz_buzz*, *indices_ofsubstring*, *spin_words*, *substitution_cipher*, *twitter* and *cut_vector*.

ChatGPT-generated sources are worse in *twitter* and *vector_distance*. Investigating these source codes we found that the higher McCC value in task *vector_distance* is due to the way it reads the input and the input is validated. The higher NOS value in task *twitter* is due to the fact that ChatGPT generated a `main` method in order to test the solution while Copilot did not.

Copilot scored worse metric values in the remaining tasks as there was no such task where one metric was better for one model and another metric was better for another model. In task *substitution_cipher* the metric NLE is 3 times higher for Copilot, which is due to embedded for-loops. ChatGPT used an unordered_map while Copilot used for-loops resulting in higher NLE and higher complexity. For task *spin_words* two metrics, NOS and McCC were worse in Copilot-generated source. In this case, both McCC and NLE higher values were caused by the fact that Copilot used for-loops to reverse a string while ChatGPT used `std::reverse` calls. For task *indices_of_substring*, the McCC and NLE metrics were twice as bad. Once again, Copilot generated code that compares substring meanwhile ChatGPT utilized the find method. For task *fizz_buzz* the LLOC metric was worse, which was caused by the additional curly-brace pairs in Copilot's code. For task *find_pair* the

TABLE 3. Static analyzer result for C++ and Java (G = ChatGPT, P = Copilot).

Task	C++										Java									
	CodeSmells		LLOC		NOS		McCC		NLE		CodeSmells		LLOC		NOS		McCC		NLE	
	G	P	G	P	G	P	G	P	G	P	G	P	G	P	G	P	G	P	G	P
basement	0	0	20	16	20	15	4	3	2	2	0	0	17	17	11	10	3	3	2	2
bouncing_balls	0	0	21	17	20	15	2	2	1	1	0	1	23	18	16	14	2	2	1	1
bowling	8	7	48	45	50	45	10	9	4	4	0	0	57	39	38	25	13	7	3	3
camel_case	2	3	20	22	20	21	4	4	2	3	0	3	16	26	11	18	2	5	1	3
coin_sums	7	6	19	14	17	12	1	1	0	0	0	0	20	20	14	13	1	1	0	0
cut_vector	2	3	27	50	29	55	5	7	2	2	1	0	28	34	21	26	3	5	2	2
dice_game	2	0	9	16	8	17	2	4	1	3	0	0	20	20	15	14	4	4	3	3
find_pair	0	2	17	34	19	32	4	4	3	2	0	0	20	29	13	25	3	6	2	3
fizz_buzz	7	3	12	21	15	18	5	4	1	1	0	0	18	19	10	10	5	5	1	1
fuel_cost	0	1	11	13	10	12	2	2	1	1	1	0	16	20	8	15	2	3	1	1
gcd	3	0	12	17	10	14	2	2	1	1	0	1	16	19	9	12	2	5	1	2
indices_of_subtring	1	0	19	18	18	14	3	6	1	2	1	0	18	23	10	17	2	4	1	2
leaders	1	2	21	22	22	23	4	4	2	2	1	2	20	34	13	30	3	6	2	2
luhn	4	3	24	19	25	18	5	4	2	3	0	0	25	24	16	17	5	4	3	2
mastermind	0	1	29	23	29	26	5	6	2	3	1	1	31	33	21	28	5	7	2	3
middle_character	2	3	14	14	12	11	2	2	1	1	0	0	15	17	7	8	2	2	1	1
paired_digits	0	0	16	17	16	16	3	3	2	2	0	0	16	21	10	13	3	4	2	2
shopping_list	2	1	19	14	19	13	3	2	1	1	0	2	16	25	11	21	2	4	1	1
snow_day	1	0	21	17	21	17	2	2	1	1	0	1	15	19	9	13	1	2	0	1
solve_boolean	1	2	30	30	33	32	9	10	2	2	2	1	30	27	22	17	11	7	3	3
spin_words	3	4	27	36	26	43	5	9	3	4	0	3	21	24	12	15	3	4	2	2
square_digits	3	4	16	17	14	14	2	2	1	1	0	1	18	18	13	10	2	2	1	1
substitution_cipher	1	2	24	26	25	27	3	4	1	3	0	1	29	19	22	13	4	2	2	1
twitter	4	4	17	12	16	9	3	3	1	1	0	1	16	18	8	9	3	3	1	1
vector_distance	2	1	30	20	30	19	6	2	1	1	0	0	15	24	10	20	2	4	1	1

metrics LLOC and NOS were worse. Investigating this source we found that Copilot used a vector for testing like ChatGPT, but it used push_back methods multiple times instead of initializing it with the generic initializer. For task *cut_vector* it was the same case. Task *dice_game* scored worse in every metric and NLE was 3 times worse. Investigating this task we found that ChatGPT used a formula to calculate the result while Copilot enumerated every possible outcome and this increased all the metrics. Note, that in this case neither of the models generated correctly functioning code but Copilot's solution is closer to a working code.

Besides the pure metrics, the number of bad coding practices and the types of these must be investigated. The number of code smells was very similar for both models. The most common code smell was using magic constants. In the solutions generated by ChatGPT, it occurred 25 times, while in the solutions of Copilot, it occurred 24 times.

From the validated coding smells the following smells were equally typical for both models: magic constant, narrowing conversion, string reallocation, and copying values instead of passing references. Coding smells typical for ChatGPT: multiple declarations in one line, not using braces for single statements. Coding smells typical for Copilot: redundant string initialization, not using range-based for-loops, using else after return, not using empty for emptiness check, redundant boolean operator.

The mentioned coding smells are categorized as Major in SonarQube. We also took into consideration the Minor labeled smells. There were three of them in ChatGPT-generated sources. All of them suggest replacing a code part with a standard algorithm, like accumulate and transform. For Copilot, the Minor labeled smells point out that there were return statements following each other and a variable's scope could be reduced.

Taking C++ both models are similar in the aspects of metrics and coding smells too. Copilot tends to generate more coding smells specifically that relate to modern C++ code.

2) JAVA RESULTS

Similarly to C++, we only discuss tasks where the models are significantly different. One of the tasks where ChatGPT scored worse is *substitution_cipher*. The NOS, McCC, and NLE metrics are worse for ChatGPT. Investigating the source code, ChatGPT generated a Map for characters meanwhile Copilot used single methods for look-up. This affects the above-mentioned metrics as there are more statements and additional loops. Another task where we investigate the code is *bowling*. In this case, ChatGPT generated a utility function for the task which led to the increased McCC as they usually include loops.

Copilot-generated tasks got worse metric values for many tasks and not only for one metric. Investigating the source code we found that for *vector_distance*, *shopping_list*, *fuel_cost*, and *cut_vector* the difference is due to testing. ChatGPT generates fixed tests while Copilot reads values from console resulting additional code e.g. for reading vectors.

In the remaining examples LLOC, McCC and NOS were higher for Copilot. Investigating the code we found that all the tasks are similar, Copilot tends to generate more iterations than ChatGPT. It uses iterations instead of using built-in iterations and methods.

Paying attention to the validated code smells again, there was no code smell which was typical for both models. ChatGPT tends to have the following code smells: using synchronized class if not needed, not using interface or abstract return types, and ChatGPT-generated one example where an unused import was present. Copilot tends to have the following code smells: misleading method name that matches the class name, not using string builder thus making the code less effective, bad naming convention for variables, having unused parameters and in one example Copilot introduced a bug by using a method overload that generates a temporary object.

For Java, we conclude that Copilot generates larger and more complex code which could be replaced by improved algorithmic solutions and function calls.

Based on these results we can answer **RQ 1: How does LLM-generated source code score in terms of source code quality?** LLMs perform quite well in terms of static analysis. They only occasionally introduce bad smells or bugs but the generated code passes the SonarQube quality gates mostly with the best score.

E. HUMAN EVALUATION

In the human evaluation, 5-5 developers were interviewed for each of the two programming languages. These developers had industry experience ranging from 5 to 20 years. We developed a simple web application for performing the manual evaluation. During the inspection, the developers were required to read the task text used to generate the source code. Once they understood the task, the generated sources were presented side-by-side for both models, with the order randomly swapped to prevent the developer from identifying which model was used. The developers were then asked to rate four properties of each source code on a [-2;2] interval, excluding 0. We chose this interval because it does not allow the developer to remain neutral, and the plus and minus values represent liking and disliking, respectively. The properties that had to be scored were the following (with their instructions towards the developers):

- First impression: Score the source code according to your first impression. There are no special aspects given that affect this score what you should consider. It is solely your impression. Scale it from −2 to 2.
- Readability: Score the source code according to its readability. Readability refers to your experience during reading the code, while you are trying to understand the various steps and conditions. How clear the purpose of a variable or a method is. How easily you can follow the flow of the code. Score it from −2 to 2.
- Usability: Score the source code according to its usability. Usability refers to your experience during interacting with the source code. It involves writing tests, including it in your source base, or using it as a black box program. It should measure how satisfied you are after using this source code. Score it from −2 to 2.
- Modifiability: Score the source code according to it's modifiability. Modifiability determines how hard it is to change the code, to add or remove functionality. Score it from −2 to 2.

Besides these properties, the developers had to decide whether they accepted the source code or not. The possible values were "Strong reject", "Weak reject", "Weak accept", and "Strong accept". These string values are mapped to the already used [−2;+2] interval where "Strong reject" refers to -2 and "Strong accept" refers to +2. Zero value is not allowed either. After scoring all the values, the developers had to decide which of the shown source code was the better one.

As the previous sections presented the quality of the generated code, this section evaluates how that code was considered by experts. Table 4 shows the developer evaluation results for C++ and Java. The values are summed scores[7] for every property, each scoring from −2 to +2, therefore, with 5 developers for each language the lowest score is −10 (marked with red in the table) and the highest is +10 (marked green). The developers not only had to score based on 5 properties, but had to decide which code was better (G/P column). The negative values mean that ChatGPT was better according to the experts while the positive values denote the cases when Copilot was better.

1) C++ RESULTS

For C++, the results in Table 4 show that ChatGPT is more welcome than Copilot. The **First Impression** for ChatGPT on average over the tasks and developers is 5.8 with a 3.7 standard deviation while Copilot reached only 2.2 with a 4.27 standard deviation. Looking at **Usability**, the average for ChatGPT was 5.0 and for Copilot it was 2.0. The standard deviation for ChatGPT and Copilot were quite similar, 4.9 and 4.5. It shows us that both models can generate quite usable code and also hard-to-use code according to our developers. Regarding **Readability**, both models achieved quite good scores, in average ChatGPT scored 7.3 and Copilot reached 5.0. The standard deviations were 2.4 and 3.2. It shows that ChatGPT generates more readable code, slightly more frequently but Copilot's score is quite similar. **Modifiability** scores are similar too in average, 6.4 for ChatGPT with a standard deviation of 2.8 and 5.4 for Copilot with 3.2 standard deviation. The **Acceptance** has a larger difference. ChatGPT reached 5.6 in average with a standard deviation of 5.5 and 2.6 for Copilot with standard deviation of 5.8. It seems both models have accepted and rejected reviews and the large deviation shows that both models can generate very good and very bad code.

Although the pure numbers show that ChatGPT is more favorable among the developers, we investigated how these numbers would change if we included only the projects where

[7] See supplementary material referenced in Section IV.

TABLE 4. Developer evaluation results for C++ and Java (G = ChatGPT, P = Copilot, F.Impr. = First Impression, Usab. = Usability, Read. =Readability, Modif. = Modifiability, Acc. =Acceptance, G/P = ChatGPT or Copilot is better).

	C++											Java										
Task	F.Impr.		Usab.		Read.		Modif.		Acc.		G/P	F.Impr.		Usab.		Read.		Modif.		Acc.		G/P
	G	P	G	P	G	P	G	P	G	P		G	P	G	P	G	P	G	P	G	P	
basement	6	6	9	6	7	8	8	8	9	9	-5	10	10	10	10	10	10	10	10	10	10	-1
bouncing_balls	3	2	-1	-7	3	5	7	7	-1	-9	-4	9	2	0	3	8	4	9	7	2	3	-3
bowling	-3	-2	1	2	2	-2	1	-1	-2	-2	0	2	6	7	5	6	6	4	3	5	4	0
camel_case	9	5	8	4	8	3	8	8	10	8	-6	6	4	-1	5	8	-2	8	2	-1	4	-1
coin_sums	8	3	8	3	8	6	8	2	7	4	-6	9	7	5	7	9	6	9	9	10	9	-5
cut_vector	2	-7	-3	-5	8	4	3	2	-2	-6	-1	9	-3	8	0	9	4	8	4	9	1	-6
dice_game	-3	-2	-9	4	8	7	4	6	-10	5	9	8	5	6	9	9	7	9	8	7	9	-3
find_pair	8	-4	6	-7	8	0	8	-2	8	-9	-10	9	0	7	4	8	4	8	5	8	2	-5
fizz_buzz	8	3	8	4	8	5	5	7	10	5	-3	7	6	10	10	9	8	7	7	9	9	2
fuel_cost	7	4	8	4	8	8	8	6	8	6	-4	8	6	9	10	10	5	9	9	10	9	-3
gcd	5	9	8	5	8	7	5	8	9	7	0	10	-1	10	7	9	4	9	7	10	3	-9
indices_of_sub.	3	-2	-3	5	10	1	5	4	-3	7	4	9	2	7	7	9	3	9	5	9	6	-2
leaders	9	6	9	4	9	7	8	8	10	8	-5	9	-6	8	6	10	-1	8	0	9	4	-9
luhn	4	7	2	2	7	4	8	3	2	2	0	2	-5	1	-4	8	4	6	2	-1	-7	-5
mastermind	6	3	8	-2	3	8	2	7	8	-1	-4	5	3	8	5	8	4	6	3	8	3	-3
middle_char.	7	7	8	7	5	8	8	8	8	8	2	8	8	9	9	7	10	7	6	8	10	3
paired_digits	10	-3	10	-2	10	3	8	8	10	-2	-8	9	4	10	6	10	5	9	7	10	4	-6
shopping_list	7	5	6	5	8	8	8	8	9	8	-3	10	1	10	2	9	-2	9	8	10	3	-8
snow_day	7	-4	6	-8	9	5	9	3	6	-8	-9	8	4	-8	-1	7	2	3	7	-9	0	3
solve_boolean	-1	0	-1	0	2	3	-2	5	-1	0	1	3	4	0	4	5	8	2	1	-2	4	2
spin_words	9	0	8	5	8	-2	8	0	9	5	-7	10	5	10	6	10	7	9	7	10	7	-1
square_digits	8	8	8	8	9	8	8	8	9	8	-3	8	8	9	7	7	6	7	7	9	8	-7
sub._cipher	8	5	6	3	9	5	8	5	8	2	-8	4	9	9	4	6	8	2	6	8	5	-1
twitter	9	3	8	5	9	7	8	8	10	7	-6	10	8	10	8	10	7	10	9	10	8	-4
vector_dist.	8	4	8	4	8	9	8	8	9	4	-5	6	2	6	7	9	7	7	6	7	8	-1
average	**5.8**	**2.2**	**5.0**	**2.0**	**7.3**	**5.0**	**6.4**	**5.4**	**5.6**	**2.6**	**-3.2**	**7.5**	**3.6**	**6.4**	**5.4**	**8.4**	**5.0**	**7.4**	**5.8**	**6.6**	**5.0**	**-2.9**
st. devition	3.7	4.3	4.9	4.5	2.4	3.2	2.8	3.1	5.5	5.8	4.4	2.5	4.2	4.6	3.5	1.4	3.2	2.3	2.7	4.9	3.9	3.5

the models reached good enough pass rate in random testing. We did it for tasks where both models reached at least 75% pass rate, for tasks where only ChatGPT scored this level of pass rate and for tasks where only Copilot did so.

Although the average ratings improved for both models, ChatGPT still performs better in every scenario but one. Considering the tasks where only Copilot reached the minimum pass rate Copilot got better scores for modifiability. From this we can conclude, although the developers had no information about the pass rates, they upscored the functional code and using this criteria ChatGPT is still favored over Copilot.

2) JAVA RESULTS

For Java, we did the same evaluations. These results are also presented in Table 4. Firstly, **First Impression** shows a great difference. ChatGPT reached an average of 7.5 with a relatively small standard deviation of 2.5 and Copilot reachd on average 3.6 with standard deviation of 4.2. It shows that ChatGPT-generated code is mostly favored by the developers while Copilot-generated code is less preferred. Regarding **Usability**, the models are close to each other, ChatGPT scored 6.4 with standard deviation of 4.6 and Copilot reached 5.4 with standard deviation of 3.5. On average, the generated code is usable, but it is varying for both models. **Readability** also shows a great difference between the models. ChatGPT reached 8.4 with standard deviation of 1.4, which means the ChatGPT generated code is consistently readable. On the contrary, Copilot reached 5.0 with standard deviation of 3.2. It shows that Copilot generated-code might be readable, but it is less readable. Copilot generates more frequently unreadable code than ChatGPT. Taking **Modifiability**, both models vary in the same manner, as the standard deviation for ChatGPT is 2.3 and for Copilot it is 2.7. The average values are 7.4 and 5.8, which shows that the generated code can be modified quite easily, be it generated by ChatGPT or Copilot. In **Acceptance**, there is no big difference compared to C++. Both models are accepted in the same manner. ChatGPT scored 6.6 while Copilot reached 5.0. The standard deviations are 4.5 and 3.9, which means that both models generate code which is mostly accepted, but either of them can generate disliked code too.

In case of Java, we also tested if the pass ratio affects the results, thus we filtered again with the 75% pass rate. The filtered results show that ChatGPT is still favored over Copilot regarding the tasks where both models performed well on the functional testing. Not surprisingly, ChatGPT is preferred over Copilot in tasks where only ChatGPT performed well in functional tests. What is more surprising is that ChatGPT also has an advantage over Copilot in tasks where only Copilot performed well in the functional tests.

Regarding this case, the First Impression and Readability were much better for ChatGPT, which indicates that even if the code is bad, the style has a great impact on developers' opinion. Usability values were similar, so developers considered bad ChatGPT code usable too.

After the human evaluations we can answer **RQ 2: Is the generated source code accepted by experts?** The developers approved the generated source code most of the time. Developers did not consider the model-synthetized code perfect or out-of-the-box usable, but they considered the code acceptable.

F. SUMMARY

With the final evaluations we can answer **RQ 3: What aspects should be considered when choosing LLM-based generative tools?**

When choosing Large Language Model (LLM)-based generative tools, particularly for code generation, it is essential to consider a variety of aspects to ensure the model is both effective and efficient.

- **High-Level Task Interpretation:** The LLM should be capable of understanding task definitions at a sufficiently abstract level. This means that the aim is to understand and effectively respond to developer requests with minimal input, ensuring both accuracy and efficiency.
- **Functional Validity Assessment:** Before considering a model suitable, it is critical to evaluate its functional validity. This involves testing the model's output in various scenarios, including both common use cases and edge cases. While edge case testing checks the model's performance under extreme or unusual conditions, providing a larger amount of random tests shows a better image of the general functionality. This comprehensive testing ensures that the model can reliably generate functional and robust code.
- **Technical Validity:** After establishing functional validity, the next step is to evaluate technical validity. This might involve performing static analysis on the generated code to detect any code smells, bugs, or unnecessarily complex solutions. The analyzer must be configured for the actual requirements of the final environment the model would be used in.
- **Human Evaluation:** Using the results from functional and technical validity assessments, you can compare different models with a high degree of confidence. Decisions can be based on these results as models might pass the given criteria, however, developers co-working with the model might discourage the usage as the generated source code does not fit their way of thinking. Developer reviews and evaluations play a crucial role in the model selection process, providing insights into each model's strengths and weaknesses. When conducting human reviews, one of the easiest ways is to provide scoring options, although it must be on an even scale to prevent neutral results. To prevent ties on certain properties a final decision should be made to decide which model-generated source code is better.

In conclusion, when selecting an LLM-based generative tool for code generation, it is essential to consider how well it understands high-level task definitions, its performance in functional and technical validity assessments, and factors related to human evaluation like first impression, usability, and modifiability. These considerations help in choosing a tool that not only meets immediate coding needs but also integrates well into the broader development life cycle.

Although we included only the basic must-include parts, there could be scenarios where other aspects are important too. In such scenarios, evaluators of the models should decide which criteria and techniques to use for assessing the quality of the generated source code. Memory usage and time complexity are factors that could be important for specific applications, although they are not required in every evaluation.

V. THREATS TO VALIDITY

Although we tried our best, there are still a few things that must be noted which might make readers doubt.

A. MODEL SELECTION

Both models are GPT-3 based models and we could have used other models. Our main goal was to show how models could be compared properly, and from what aspects, which was done via an actual comparison. We did choose from the most popular models, the ones which were already in use in the software engineering community. Codex was trivial to use due to its plugin-like nature and ChatGPT was easy to use as it provided a handy web interface. Although the selected models are similar and in the same lineage, the training sets and techniques are different. ChatGPT is a later version, developed to be instruction-following and helpful for humans, while Codex is trained mainly for source code related tasks.

B. LANGUAGE SELECTION

One might consider why we did not include a scripting language, like Python. As we described the main aspects on language selection Python did not really fit those properties. We assume that the results would be similar using Python as there is plenty of code (training data) written in Python.

C. PROMPTING

As we could use a prompt that is designed for this purpose, we could not alter the inner workings of the models, such as randomness. Although Copilot provides a possibility to set the temperature or top_p values, the web interface for ChatGPT does not, thus in order to have a fair comparison we used default values for both. Additionally, setting the temperature might not force the model to be deterministic enough as shown by Ouyang et al. [34]. Besides setting these values, a well-known technique is to not only select the first generated value but use more, frequently marked as @1 @5 for the first and the first five values. As evaluating every task in both C++ and Java, e.g. 5 values would take too much human effort we decided to use @1 results. Otherwise selecting from @5 values or interacting with further prompting would have required human interaction which would not be objective. It is similar to interactive instructions for making the models' output better with human supervision. Results would highly depend on the supervisor's expertise and ability to instruct LLMs.

The selected benchmark was published in July, 2021 therefore there is a slight chance that ChatGPT has already seen the tasks during training as it contains information up until September, 2021. We cannot state such things from Codex as we have no insight into its real training data or dates. Although it is a possible flaw, we consider the amount of text used in the benchmarks would be greatly outnumbered by the total training corpus therefore, it would not alter the results, furthermore, in real-life usage a specific benchmark usage resolves this problem.

D. FUNCTIONAL EVALUATION

During the functional evaluation we used epsilon comparison for float comparisons. Although float values are compared with epsilon values, the benchmark did not include such values. The benchmark used in our evaluations could be changed but it was a key factor to have such a benchmark that is specially created for program generation or program synthesis.

E. TECHNICAL VALIDITY

During static analysis, we did use metrics on C++ and Java, which languages support OOP. The analyzed metrics did not include OOP-specific metrics as the tasks were way too small to properly utilize OOP.

F. HUMAN EVALUATION

We involved only 5-5 developers for both languages. To overcome the low number of developers our main goal was to have professional reviewers qualified for such a role, so we preferred quality over quantity.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a methodology for evaluating Large Language Models' (LLM) code synthesis capabilities to help developers choose the best available model. This methodology takes into account that prompting is very important for LLMs and how detailed a usual task description is. The primary concern is that the generated code works well, so it needs to be functionally tested. In addition, technical quality aspects are also important if the code is to be used in the long term. On the basis of this evaluation, a model can be chosen, but it is also worth asking the experts for their opinion.

We applied this methodology in a case study and evaluated and compared ChatGPT and Copilot on a publicly available code synthesis benchmark consisting of 25 tasks. For prompting, we used the specifications in the benchmark, which were specifically designed to test program synthesis, and accepted the first generated solution. We functionally tested the generated code with general and edge cases and found that the majority of the generated code was functionally correct. We then used static code analysis to check the technical quality and found that despite minor errors, the models generated good-quality code. Finally, we involved experts to review the solutions and their opinion supports our results. Our conclusion is that both ChatGPT and Copilot can be used for program synthesis, but based on the comparison it seems that ChatGPT is better.

We consider the proper comparison of LLMs in code generation an important task which will be inevitable during industrial software development. Based on this study we aim to create a framework in the future where LLMs can be compared in the most automatized way possible, including memory and time consumption values. Using this comparison we aim to observe the preferred models of developers among the vast amount of models. This opens the opportunity to investigate what model features are preferred providing more information to develop better models. Using a decent comparing methodology we can also investigate the effects of various prompting techniques and fine-tunings, therefore, providing knowledge for improving models.

REFERENCES

[1] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, and S. Lundberg. (Mar. 2023). *Sparks of Artificial General Intelligence: Early Experiments With GPT-4. March 2023*. [Online]. Available: https://www.microsoft.com/en-us/research/publication/sparks-of-artificial-general-intelligence-early-experiments-with-gpt-4

[2] A. Dieumegard, A. Toom, and M. Pantel, "Model-based formal specification of a DSL library for a qualified code generator," in *Proc. 12th Workshop OCL Textual Modeling* New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 61–62, doi: 10.1145/2428516.2428527.

[3] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," in *Proc. 3rd ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1976, pp. 19–31.

[4] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, and J. Teich, "An evaluation of domain-specific language technologies for code generation," in *Proc. 14th Int. Conf. Comput. Sci. Appl.*, Jun. 2014, pp. 18–26.

[5] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.

[6] P. Black, "Static analyzers: Seat belts for your code," *IEEE Secur. Privacy*, vol. 10, no. 3, pp. 48–52, May 2012.

[7] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *Proc. Int. Softw. Metrics Symp.*, Chicago, IL, USA, 2004, pp. 1–12.

[8] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proc. Extended Abstr. CHI Conf. Hum. Factors Comput. Syst.* New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 1–7, doi: 10.1145/3491101.3519665.

[9] N. Al Madi, "How readable is model-generated code? Examining readability and visual inspection of Github copilot," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.* New York, NY, USA: Association for Computing Machinery, Feb. 2023, pp. 1–5, doi: 10.1145/3551349.3560438.

[10] K. E. Someoliayi, N. Y. M. Harrand, S. Larsén, H. Adzemovic, H. L. Phu, A. Verma, F. Madeiral, D. Wikstrom, and M. Monperrus, "Sorald: Automatic patch suggestions for SonarQube static analysis violations," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 4, pp. 2794–2810, Apr. 2022.

[11] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, "Automatically generating fix suggestions in response to static code analysis warnings," in *Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2019, pp. 34–44.

[12] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? A closer look at realistic usage of SonarQube," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, May 2019, pp. 209–219.

[13] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "JuCify: A step towards Android code unification for enhanced static analysis," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1232–1244, doi: 10.1145/3510003.3512766.

[14] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, and N. Meng, "Example-based vulnerability detection and repair in Java code," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 190–201, doi: 10.1145/3524610.3527749.

[15] J. R. B. Higuera, J. B. Higuera, J. A. S. Montalvo, J. C. Villalba, and J. J. N. Pérez, "Benchmarking approach to compare web applications static analysis tools detecting OWASP top ten security vulnerabilities," *Comput., Mater. Continua*, vol. 64, no. 3, pp. 1555–1577, 2020.

[16] J. Liu and X. Zhang, "ReX: A framework for incorporating temporal information in model-agnostic local explanation techniques," 2022, *arXiv:2209.03798*.

[17] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 90–121, Jan. 1980, doi: 10.1145/357084.357090.

[18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, "Language models are few-shot learners," in *Proc. NIPS*, 2020, pp. 1877–1901.

[19] R. Thoppilan et al., "LaMDA: Language models for dialog applications," 2022, *arXiv:2201.08239*.

[20] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 649–657.

[21] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: A comparison of the program synthesis performance of GitHub copilot and genetic programming," in *Proc. Genetic Evol. Comput. Conf.* New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1019–1027, doi: 10.1145/3512290.3528700.

[22] T. Helmuth and P. Kelly, "PSB2: The second program synthesis benchmark suite," in *Proc. Genetic Evol. Comput. Conf.* Lille, France: Association for Computing Machinery, Jun. 2021, pp. 10–14, doi: 10.1145/3449639.3459285.

[23] A. Babar, "Programmer's new friend: Github copilot," *Int. J. Res. Publication Rev.*, vol. 3, pp. 2721–2725, Nov. 2022.

[24] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1219–1231, doi: 10.1145/3510003.3510203.

[25] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2339–2356.

[26] O. Asare, M. Nagappan, and N. Asokan, "Is Github's copilot as bad as humans at introducing vulnerabilities in code?" *Empirical Softw. Eng.*, vol. 28, p. 129, Jan. 2023.

[27] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 754–768.

[28] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *Proc. IEEE 22nd Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Oct. 2022, pp. 71–82.

[29] R. Khoury, A. R. Avila, J. Brunelle, and B. Mamadou Camara, "How secure is code generated by ChatGPT?" 2023, *arXiv:2304.09655*.

[30] *How to Get Codex to Produce the Code You Want!* Accessed: Apr. 2, 2024. [Online]. Available: https://microsoft.github.io/prompt-engineering/

[31] P. Denny, V. Kumar, and N. Giacaman, "Conversing with copilot: Exploring prompt engineering for solving CS1 problems using natural language," in *Proc. 54th ACM Tech. Symp. Comput. Sci. Educ.* New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 1136–1142, doi: 10.1145/3545945.3569823.

[32] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," 2023, *arXiv:2303.07839*.

[33] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota, "Source meter sonar qube plug-in," in *Proc. IEEE 14th Int. Work. Conf. Source Code Anal. Manipulation*, Sep. 2014, pp. 77–82.

[34] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "LLM is like a box of chocolates: The non-determinism of ChatGPT in code generation," 2023, *arXiv:2308.02828*.

ZOLTÁN SÁGODI is currently pursuing the Ph.D. degree.

Despite this being the first year of the Ph.D. degree, he has already been working in research during his M.Sc. and even B.Sc. studies. His Ph.D. topic is detecting vulnerabilities and faults in source code via static analysis and AI application. He is also with the Department of Software Engineering, University of Szeged. Besides his research tasks, he takes his part in education. This means multiple courses and in his freetime, he takes effort into automatizing many of the educational tasks (e.g., creating and correcting exams and preparation for the courses).

ISTVÁN SIKET received the Ph.D. degree in computer science, in 2011. He is currently an Assistant Professor with the Department of Software Engineering, University of Szeged. His research interests include source code analysis, measurement, quality assurance, and bug detection. He has been participating in several research and development projects related to source code analysis and quality assurance.

RUDOLF FERENC received the Ph.D. degree in computer science from the University of Szeged, in 2005, and the Habilitation degree, in 2015.

He is currently an Associate Professor and acting as the Head of the Department of Software Engineering, University of Szeged. His research interests include static code analysis, metrics, quality assurance, design pattern and antipattern mining, and bug detection. He leads the Static Code Analysis Group, which develops tools for analyzing the source code of various languages. These tools calculate code metrics and detect coding issues and duplications. He has more than 100 publications in these fields with over 2000 citations. He is leading several research and development projects, which are related to quality assessment, improvement, and architecture reconstruction of software systems for major banks and software development companies in Hungary. He has been serving as the Program Co-Chair and a Program Committee Member at the major conferences in this field (ICSE, ICSME, ESEC/FSE, SANER, CSMR, WCRE, ICPC, SCAM, and FASE), since 2005.