The End of an Era: Can AI subsume Software Developers? Comparing the Effectiveness of ChatGPT and Co-Pilot for Python Code

Author(s)
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

Abstract*—Artificial intelligence (AI) has become increasingly popular in software development to automate tasks and improve efficiency. AI has the potential to help while developing or maintaining software, in the sense that it can produce solutions out of a textual requirement specification, and understand code to provide suggestion on how a new requirement could be implemented. In this paper, we focus on the first scenario. Two AI-powered tools that have the potential to revolutionize the way software is developed are OpenAI's ChatGPT and GitHub's Copilot. In this paper, we used LeetCode, a popular platform for technical interview preparation and personal upskilling (self-learning), to evaluate the effectiveness of ChatGPT and Copilot on a set of coding problems, along with ChatGPT's ability to correct itself when provided with feedback. The analysis of the effectiveness can lead to various conclusions, such as on if these solutions are ready to take over coding roles, and to what extent several parameters (difficulty and quality requirements) influence this result. Solutions have been generated for 60 problems using ChatGPT and Copilot, for the Python programming language. We investigated the performance of the models, the recurrent kinds of errors, and the resulting code quality. The evaluation revealed that ChatGPT and Copilot can be effective tools for generating code solutions for easy problems while both models are prone to syntax and semantic errors. Small improvements are observed for ode quality metrics across iterations, although the improvement pattern is not consistently monotonic, questioning ChatGPT's awareness of the quality of its own solutions. Nevertheless, the improvement that was found along iterations, highlights the potential of AI and humans, acting as partners, in providing the optimal combination. The two models demonstrate a limited capacity for understanding context. Although AI-powered coding tools driven by large language models have the potential to assist developers in their coding tasks, they should be used with caution and in conjunction with human coding expertise. Developer intervention is necessary not only to debug errors but also to ensure high-quality and optimized code.

***Index Terms*—OpenAI ChatGPT, GitHub Copilot, code quality**

I. Introduction

As artificial intelligence (AI) continues to advance, there has been a growing interest in the development of AI-powered tools. Two such tools that have gained significant attention are GitHub Copilot and OpenAI's ChatGPT. Copilot is a tool that can suggest code snippets and completions as developers write code, while ChatGPT can generate human-like responses to natural language prompts.

Both tools share the same neural network architecture created by OpenAI, which enables the understanding of natural language (Copilot is powered by the Codex model, a descendant of GPT) and are examples of Large Language Models (LLMs). One revolutionary aspect of these models is the use of the Transformer architecture. The self-attention mechanism in such architectures enables focusing on specific parts of the input when making predictions. These models came into the spotlight after the release of Google's seminal paper: "*Attention Is All You Need*" [1].

These tools have the potential to significantly speed up software development: The GitHub Next team, surveyed 95 developers split into two groups: with and without access to Copilot.[1] They observed that Copilot enabled a higher and faster task completion rate. However, there were also concerns about the quality of the generated code. As with any global phenomenon, skepticism has emerged for these models, regarding their accuracy, bias, insufficient data protection, ethical use, etc. Public media have also highlighted the risk of AI affecting jobs; software development being one such industry.[2]

To better understand the capabilities and limitations of these AI models, it is important to evaluate them in real-world scenarios, or contexts that are resembling real-world scenarios. To this end, we assess ChatGPT and Copilot using problems from LeetCode[3], a popular online platform for practicing kinds of coding problems commonly asked in technical interviews [2]. LeetCode problems are based on real-world scenarios in topics such as algorithms and data structures and cover a wide range of difficulty levels. Despite some criticisms regarding their divergence from day-to-day coding tasks, these challenges provide a structured and diverse set of problems, enabling the evaluation of LLMs in problem-solving scenarios

[1] https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness

[2] https://www.businessinsider.com/chatgpt-jobs-at-risk-replacement-artificial-intelligence-ai-labor-trends-2023-02

[3] https://leetcode.com

commonly encountered in technical interviews. This approach offers insights into how LLMs might perform in real-world coding tasks, a crucial aspect of software development.

Moreover, we focus on the Python programming language, given its widespread popularity and significant representation in LLM training datasets. This choice allows for an analysis of the capabilities and limitations of LLMs in a commonly used programming environment. Since ChatGPT is an interactive model which stores chat history and can process input provided by humans to modify its answers, we analyze the extent to which ChatGPT can improve its code solutions when given feedback. Finally, we consider for each problem a range of solutions that are generated in each iteration of the dialogue, and then we proceed with analyzing these solutions for code quality and performance metrics.

In summary, this paper presents a comprehensive study on the application of LLMs in the realm of software development. The core contribution lies in the exploration and analysis of the effectiveness and limitations of the investigated models in generating and improving Python code. More importantly, the analysis relies on a thorough qualitative assessment of the generated output. This study not only demonstrates the practical utility of LLMs in software engineering tasks but also helps lay the groundwork for future research in this rapidly evolving field.

The rest of the paper is organized in the following way: Section II discusses related work; in Section III we present the design of our study. The results of our study along with their discussion are reported in Sections IV and V, respectively. Section VI reports the threats to the validity of our study; finally, we conclude the paper in Section VII.

II. Related Work

ChatGPT is capable of responding to a wide range of questions and prompts, which include source code generation if required. Because of the short time that this model and tool existed, to the best of our knowledge, their is a limited amount of studies about ChatGPT in the context of software development. But in the last months OpenAI just released its successor, GPT-4 (on March 2023), and other big companies introduced their own language models [4] [5], so their validation and study are of vital importance.

A. Copilot

After GitHub Copilot's first appearance, a number of studies focusing on the performance of Copilot emerged. Due to the trending aspect of such technologies, there seems to be an increasing number of papers from 2022. The main focus of the existing studies is on the correctness of Copilot, where either specific code problems are used [3], [4] or professional programmers are being compared against Copilot [5]. The results of these studies showed that although Copilot can help with the performance and the time that a developer needs to complete a specific task, in most cases its not enough or the quality is inferior. In the first study 70% of the solutions provided by Copilot were only partially correct or entirely incorrect.

Furthermore, a number of studies focused on the usability of such models [6], [7]. Developers appear to consume more time to finish a task that Copilot answered incorrectly, because they have to understand the generated code first, and then start debugging. A similar study by Arghavan et al. [8] taking into account the difficulty of using Copilot, encourages its use. They authors believe that if correctly used, Copilot can boost the performance of the developers. Finally, the generated code's security was studied by Hammond et al. [9] revealing that approximately 40% of the generated code can be subject to vulnerabilities.

B. ChatGPT

For ChatGPT the number of published studies is smaller. Scoccia [10] studied the developers' perception of ChatGPT, taking into account the user experience, the trust in the generated code, prompt engineering, and the impact of these models in software development. These four categories appear to be the main topics that the developers tend to focus on regarding ChatGPT or similar models. According to Ahmad et al. [11] ChatGPT could be used to help in the architecture stage of software design, but more studies are required. Moreover, another area that ChatGPT can help with, is the domain of unit testing. Guilherme and Vincenzi [12] used 33 programs in order to generate unit tests with ChatGPT, and they found out that the results were very good and in line with dedicated tools that already exist for unit test generation.

Finally we note that a similar study conducted by Nguyen and Nadi [13] looked at the correct code suggestions of Copilot for 33 LeetCode problems in 4 different languages (Python, Java, JavaScript, C). Our study, even though it focuses only on Python, examines a larger body of problems, employes the newer ChatGPT model, and also looks at incorrect solutions, as well as the quality of the proposed code.

III. Case Study Design

In this section, we present the study design, by providing information about the research questions, the data collection process, and the employed statistical analysis. The case study was designed and reported based on the guidelines provided by Runeson et al. [14].

A. Research Questions

To study ChatGPT's and Copilot's ability to solve coding problems on LeetCode, we have formulated and set the following research questions:

- **RQ1:** What is the effectiveness of ChatGPT and Copilot in solving LeetCode problems in Python?
- **RQ2:** Can ChatGPT effectively act on feedback to debug Python code?
- **RQ3:** To what extent can ChatGPT improve the performance and quality of Python code?

[4] https://blog.google/technology/ai/google-palm-2-ai-large-language-model

[5] https://ai.meta.com/llama/

As mentioned before, we have chosen to focus primarily on Python due to its widespread popularity and extensive representation in the training datasets of LLMs. Python's usage in diverse fields from web development to data science ensures a rich variety of code examples and scenarios for analysis. Also, Python's syntax and community conventions make it an ideal candidate for examining the capabilities of LLMs in a real-world programming context.

In RQ1 we aim to find the effectiveness of the models in solving LeetCode problems. This will provide some insight in the ability of ChatGPT and Copilot to write code for close to real-word problems. To achieve this, we selected a number of LeetCode problems, fitted them to the models, and used the proposed solution without any changes.

Since ChatGPT's main advantage is the "communication" that one can have with the model, in RQ2 we try to investigate the ability of the model to correct itself. For that, we allow ChatGPT to rectify any incorrect solutions with the help of feedback on errors in the form of stack traces or information around failed test cases. This choice stems from our desire to investigate LLMs' ability to interact with automated feedback mechanisms, such as compiler errors or test case results. This approach mirrors real-world development environments where automation tools often guide code refinement. It also allows us to reflect on the potential of tools like AutoGPT, facilitating an automated loop of code improvement before developer intervention.

Finally, we take the communication with the model one step further and we ask it to improve the code quality and performance of its correct solutions in order to understand its capacity to improve of preexisting code. By following this design, our study juxtaposes two distinct avenues for code improvement: automatable mechanisms, represented by compilers and test cases, and non-automatable mechanisms, such as common development goals and practices (e.g., overall quality improvement). This dual approach allows us to assess the versatility of LLMs in both structured and more fluid, human-centric coding scenarios.

By investigating these questions, we aim to shed light on the potential of language models like ChatGPT in enhancing the coding experience for programmers. Furthermore, the findings can contribute to the broader understanding of the capabilities and limitations of LLMs in the context of algorithmic problem-solving.

B. Case Selection

Objectively assessing the capabilities of language models to carry out programming tasks is challenging, considering that they stem from supervised learning and as a result might perform better in tasks that resemble the dataset on which they have been trained. To alleviate any bias and test them on problems with varying degrees of difficulty, we evaluate the performance of the models on 60 randomly selected problems from LeetCode, which is a popular online platform for programming learning and assessment. To reduce any bias in the problem selection we used the random selector offered by the platform (namely the option "*Pick One*") for each of the three difficulty categories. Thus we randomly selected 20 problems from each of the categories (Easy, Medium, and Hard) bringing the total number of problems to 60.

Orthogonal to our efforts to mitigate biases, the choice of LeetCode can be a subject of debate, particularly concerning its representation of real-world coding scenarios. While LeetCode primarily prepares developers for coding interviews, the platform's challenges encompass a spectrum of problems that developers frequently encounter. Our rationale is that, despite their interview-oriented design, these challenges offer a valuable proxy for gauging LLMs' effectiveness in typical development tasks, reflecting a range of complexity and problem-solving skills pertinent to everyday programming.

C. Data Collection

In order to collect the appropriate data for RQ1, for each problem, we retrieved its description and prompted each model (ChatGPT and Copilot). Each prompt followed the following format:

> *Write the code that solves the following problem using the Python programming language.*
> <problem description from LeetCode>

Sometimes an additional interaction with the model was needed, in order to specify the parameters of the method signature. This was necessary to not alter the generated code. The final model-generated solution was then submitted, without any alterations, to LeetCode's submission system. We recorded the status (i.e., pass or fail) and the message (in case of failure). We should note that for each problem a new chat (conversation thread) was used to avoid interference of chat history from previous problems with new responses.

Regarding RQ2 and RQ3, once again we had to "communicate" with the AI model, but only with ChatGPT, since it is the only one that has the ability to keep track of a specific conversation. In order to collect our data we initiated a dialogue with the ChatGPT client, as described in the next figure and explained below.

1) First we ask ChatGPT for a solution to the problem. This includes providing the problem description as copied from LeetCode, an instruction specifying that a solution in Python is to be generated, and a function prototype that this solution has to follow in order to match the input format required to submit to LeetCode. The solution generated by ChatGPT will be copied into the LeetCode console and submitted. If the solution is accepted, we move to step 3. Else, we move to step 2.
2) LeetCode provides information about the error encountered with the submitted solution. The stack trace leading to the error in case of a runtime error, and the failing testcase in case of wrong answers and time limit exceeded errors is displayed. We provide ChatGPT the error details available and ask it to correct its code. We then copy the revised version of the solution that ChatGPT provides into the LeetCode console and submit

it again. If the solution is accepted, we move to step 3. Else if step 2 has been repeated 5 times and the solution still fails, we terminate the dialogue and mark the problem as failed. Else, we repeat step 2.

3) At this point, we have a solution that is accepted by LeetCode. We note the runtime and memory performance percentile statistics provided by LeetCode on the accepted solution page. We re-upload the same solution twice and take note of the performance statistics of each iteration in order to later calculate the average values. If this step has already been repeated 5 times, we terminate the dialogue here. Else, we ask ChatGPT if it can improve the time and space complexity of its solution. If it says that no further improvement is possible, we terminate the dialogue. Else, we upload its 'improved' solution to LeetCode. If it is accepted, we repeat step 3. Else, we terminate the dialogue.

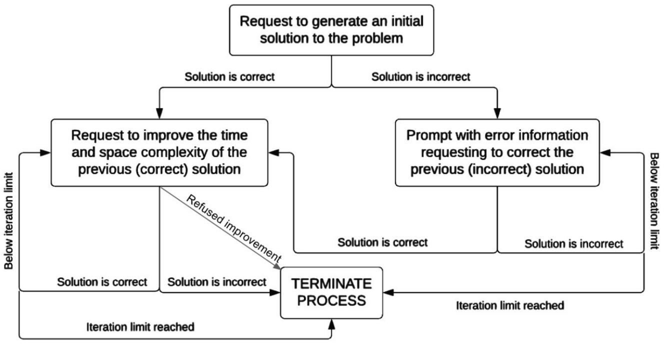


Fig. 1. Diagram for the dialogue conducted with ChatGPT

Moreover, in order to collect the data for the code quality and performance metrics, for the RQ3, we had to make some extra steps. Regarding the code quality metrics the complexity analyser Lizard [15] was used to automate these calculations. These metrics primarily capture the visual complexity of the code, providing an indication of how maintainable or readable it is rather than how it performs [16]. The metrics that we used were the following:

1) *Cyclomatic Complexity*: his is a measure of the number of linearly independent paths in a program [17]. This correlates directly to the number of decision points in the program. Low cyclomatic complexity is considered an indicator of high code quality
2) *Token Count*: The total number of operators and operands in a program.
3) *Lines of Code*: The number of lines of code in the function.

Finally, in order to take into account the code performance we used the following two metrics:

1) *Time Complexity*: The time taken for the program to execute is known as its runtime. In our paper, the runtime can vary for each function depending on the provided input. LeetCode provides the runtime in milliseconds for each submitted solution over its set of test cases, along with the percentile of submissions it outperforms. However, it is important to note that these numbers can be unreliable, as indicated by various forums [6] [7] and the standard deviation in results. To obtain a more accurate assessment of performance across iterations, we have manually evaluated the best, average, and worst-case time complexity of each solution.
2) *Space Complexity*: The amount of memory used by a program to store data during execution is called its memory usage. Similar to runtime, the space usage can vary for each function depending on the input. LeetCode also provides the memory usage in megabytes for each submitted solution over its set of test cases, along with the percentile of submissions it outperforms. However, like the runtime metric, these numbers can be unreliable. To address this, two researchers have manually evaluated the best, average, and worst-case space complexity of each solution to better capture the change in performance throughout iterations. Disagreements between researchers were discussed and a consensus was reached.

D. Data Analysis

For RQ1, we employed a combination of quantitative and qualitative analyses to evaluate the effectiveness of Copilot and ChatGPT in Python code generation. This includes examining success rates in code compilation and problem-solving, complemented by a thematic analysis of the types of errors and their frequencies.

For RQ2, we first continued our interaction with ChatGPT and completed the data collection by generating all 'improved' solutions for the selected problems. Next, we employed a similar quantitative and qualitative analysis to reason on the ChatGPT's ability to act on non-requests of non-functional nature.

Finally, for RQ3, after calculating the Cyclomatic Complexity, Token Count, and Lines of Code for each correct solution (with at least two correct solutions), the data was analyzed to identify patterns and trends across iterations. Each problem had between 2 and 6 iterations available for analysis. The following comparisons were made considering the correct solutions for each eligible problem:

1) *Last versus first value*: This metric compares the initial and eventual correct solution for each problem. There are three possible scenarios for the eventual solution: when ChatGPT determines that no further improvement is possible; when an 'improved' solution no longer satisfies all the LeetCode test cases, making the last correct solution the eventual solution; and the last solution when the iteration limit is reached and the dialogue is terminated.
2) *Best versus first value*: This metric compares the initial and best ranking solutions obtained across all iterations

[6] leetcode.com/discuss/general-discussion/136683/different-run-time-with-same-code

[7] leetcode.com/discuss/general-discussion/556815/leetcode-run-time-memory-usage-discrepancy

for each problem. It is important to note that the initial solution itself may also be the best. During the initial data collection, it was observed that in some cases, subsequent iterations deteriorated in one or more metrics instead of consistently improving. Therefore, this comparison serves as a measure of the maximum potential improvement.

For Cyclomatic Complexity, Token Count and Lines of Code, the mean difference in the last-first and best-first value pairs across solutions is calculated. However, its magnitude can be hard to interpret, since it represents the difference in each metric across iterations of solutions without considering their absolute values, which can be vastly different between different problems. For example, a change in Cyclomatic Complexity from 10 to 8 is different from a change from 4 to 2, even though the absolute difference is the same. So, in addition to calculating the central tendencies, Wilcoxon Signed-Rank tests are performed to determine whether there was a statistically significant decrease in the metrics from the first iteration to the last or best. This nonparametric test is chosen because we cannot assume a specific distribution for the data points (the boxplots show certain skews), and the sample data consists of paired values (metrics from two solutions of the same problem). This way, the values themselves of these metrics in the selected iterations will be considered rather than simply their differences. The null hypotheses in context of each of the metrics assume that the last/best iteration's solution is not significantly different from that of the first. The corresponding alternative hypotheses assume the last/best iteration's value for each metric is lower than that of the first. In the case of Time and Space Complexities, only the average case is used in this analysis, since the best case occurs in very few scenarios and the average case, which is the most common case, is found to largely coincide with the worst case.

IV. Results

A. Effectiveness of ChatGPT and Copilot

In Table I, we present the summary of the performance for both ChatGPT and Copilot models grouped by problem difficulty level (easy, medium, hard), programming language, and error. To provide a more complete and comparative picture of the evaluation, we visualize the status of each solution in Fig. 2. A colored circle indicates a correct solution for the corresponding problem.

The main takeaway message is that both models seem well-versed in easy problems while ChatGPT seems to be a bit more capable. Both ChatGPT and Copilot make notably more Logical errors, than Runtime of Time Limit. It can be concluded that when the models understand the provided problem it is very likely that they will provide a correct solution.

TABLE I
Performance of ChatGPT and Copilot

Facet (N)	Copilot		ChatGPT	
correct solutions per problem level				
easy (20)	14	(70.0%)	16	(80.0%)
medium (20)	10	(50.0%)	14	(70.0%)
hard (20)	7	(35.0%)	10	(50.0%)
incorrect solutions per error type				
Logical (60)	18	(30.0%)	14	(23.0%)
Runtime (60)	5	(8.0%)	4	(7.0%)
Time Limit (60)	6	(10.0%)	2	(3.0%)

Looking at the errors of the generated code, most were semantic in nature, rooted in either logical or performance (Time Limit) issues. But we note that solving such issues does not imply having a correct solution. In a nutshell, these results inform the starting point where engineers can progress toward a reliable and efficient solution. For example, see the problem statement for *Recover the Original Array*[8]:

> A 0-indexed array consisting of n positive integers, was splitted into two by a given integer k.

The main requirement is to recover all three arrays (original and the two after the split) by getting only the numbers that existed in all of them. The problem that ChatGPT faced was a runtime one, i.e., there was a possible division by 0. The first three lines of the provided solution were the following:

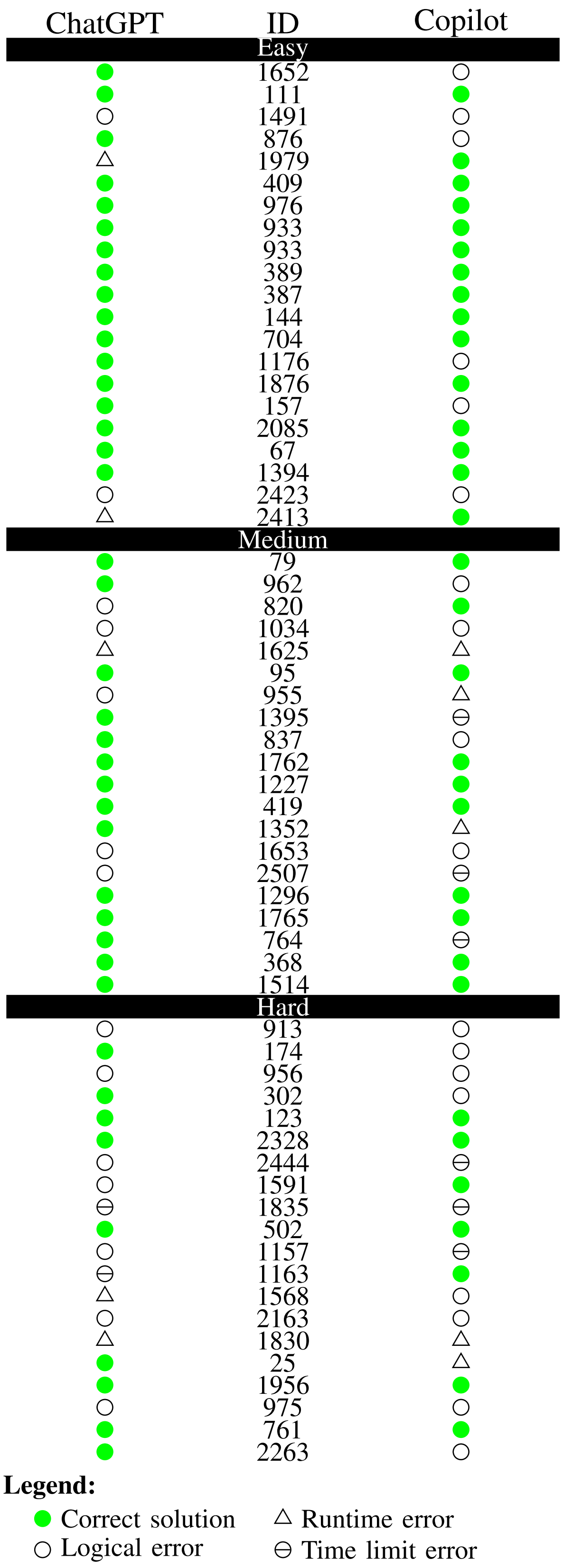
```
Arrays.sort(nums);
int n = nums.length / 2;
int k = (nums[nums.length-1]
            - nums[0]) / (2*n-2);
```

When the original array (`nums`) contains two numbers, the denominator becomes zero. This shows the importance runtime issues could play in a given scenario that the model did not 'think about' but a human could have foreseen. We should note that the comparison between human and machine is something we did not delve into for this study, but Imai [5] discusses some early evidence.

B. Act on feedback

After the interaction with ChatGPT, a total of 47 out of 60 problems had at least one correct solution, i.e., seven more solved problems compared to the the results presented in RQ1. Table II shows the distribution of what halted the solution improvement process for each problem solved successfully. The majority of solutions (31 out of 47) were terminated due to a wrong answer, indicating a lack of enough reasoning to establish what is a correct solution. This is further supported by the seven cases where ChatGPT just kept changing the code. Still, in nine occasions, ChatGPT claimed to have reached an optimal solution and refused future improvement. But given the previous results, there is less credibility to support reasoning. Overall, this highlights the need for human oversight in the development process, especially when utilizing such models for code generation and optimization.

[8] https://leetcode.com/problems/recover-the-original-array



ChatGPT	ID	Copilot
	Easy	
●	1652	○
●	111	●
○	1491	○
●	876	○
△	1979	●
●	409	●
●	976	●
●	933	●
●	933	●
●	389	●
●	387	●
●	144	●
●	704	●
●	1176	○
●	1876	●
●	157	○
●	2085	●
●	67	●
●	1394	●
○	2423	○
△	2413	●
	Medium	
●	79	●
●	962	○
○	820	●
○	1034	○
△	1625	△
●	95	●
○	955	△
●	1395	⊖
●	837	○
●	1762	●
●	1227	●
●	419	●
●	1352	△
○	1653	○
○	2507	⊖
●	1296	●
●	1765	●
●	764	⊖
●	368	●
●	1514	●
	Hard	
○	913	○
●	174	○
○	956	○
●	302	○
●	123	●
●	2328	●
○	2444	⊖
○	1591	●
⊖	1835	⊖
●	502	●
○	1157	⊖
⊖	1163	●
△	1568	○
○	2163	○
△	1830	△
●	25	△
●	1956	●
○	975	○
●	761	●
●	2263	○

Legend:
● Correct solution
○ Logical error
△ Runtime error
⊖ Time limit error

Fig. 2. Distribution of problems solved per language and model

Looking at the errors related to all problems, including those that were successfully solved at some point, we collected a total of 119 incorrect solutions. In Table III, we show their distribution according to the same type of errors used in RQ1. The results corroborates that ChatGPT's main struggle is in properly composing the logic required to accurately address the problem statement.

TABLE II
DISTRIBUTION OF THE THREE CASES IN THE IMPROVEMENT PROCESS

Case	Frequency
Optimal solution reached	9
Iteration limit reached	7
Termination due to wrong answer	31

TABLE III
DISTRIBUTION OF ERROR TYPES ACROSS ALL THE WRONG SOLUTIONS

Type of error	Frequency
Logical	93
Runtime	14
Time Limit	12

C. Improve performance and quality

1) Code Quality Metrics: From the boxplots in Figure 3, we can see that the change in values of cyclomatic complexity, token count and lines of codes of solutions between the first and the last iterations is distributed in both the positive and negative range.

In Table IV it can be seen that while the median change in all three metrics is 0, the mean difference in values between the first and last iterations is positive for two out of the three metrics, suggesting an overall increase in token count and lines of code from the first iteration's solution to the last's. The Best - First iteration difference, by definition, is the first value of each metric subtracted from the lowest value out of all iterations, and hence cannot be positive.

The mean magnitude of difference in cyclomatic complexity and lines of code between the first and best iteration in our sample data is small but negative, however we observe a large

TABLE IV
MEAN AND MEDIAN FIRST AND LAST OR BEST ITERATIONS ACROSS ALL PROBLEMS

Metric	Best - First difference		Last - Fist difference	
	Mean	Median	Mean	Median
Cyclomatic Complexity	-0.78	0.00	-0.22	0.00
Token Count	-10.47	0.00	1.91	0.00
Lines of Code	-1.25	0.00	0.13	0.00

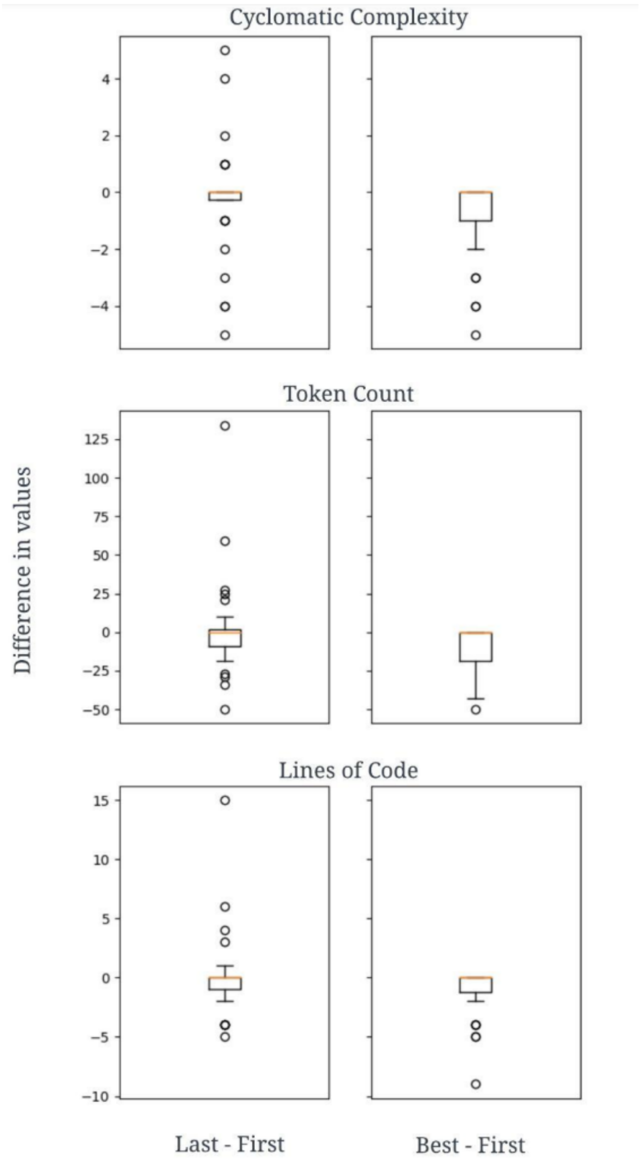


Fig. 3. Boxplot of the distribution of the differences

average decrease of 10.47 lines of code between the first and best iterations' solutions.

The null hypotheses assume for each metric that the last/best iteration's solution is not significantly different from that of the first. The corresponding alternative hypotheses assume the last/best iteration's value for each metric is lower than that of the first. The resulting p-values for the tests comparing the first and last iterations were 0.26, 0.33, and 0.31 for the cyclomatic complexity, token count and lines of code cases respectively, each case of which is higher than the significance level. Consequently, the null hypotheses are not rejected, indicating no significant decrease in cyclomatic complexity, token count or lines of code between the number of iterations.

The resulting p-values in the tests comparing the first and best iterations were less than 0.01 for cyclomatic complexity, token count and lines of code respectively, which in each case are below the significance level, indicating that the null hypothesis is rejected and that there is evidence to support that the difference in cyclomatic complexity, token count and lines of code between the first and best solutions is statistically significant.

TABLE V

PROPORTION OF PROBLEMS WHERE THE BEST VALUE OF EACH METRIC IS FOUND IN THE FIRST SOLUTION, LAST SOLUTION OR NEITHER

	First is Best	Last is Best	Neither First Nor Last is Best
Cyclomatic Complexity	0.6875	0.7500	0.0625
Token Count	0.5625	0.5313	0.1875
Lines of Code	0.5938	0.6875	0.1250
Average time complexity	0.8438	0.9063	0.0313
Average space complexity	0.9063	0.8750	0.0000

2) Performance Metrics: The p-values obtained with the null hypothesis being that there is no significant difference between the time or space complexities of the first and last iterations of solutions are 0.23 and 0.70 respectively, suggesting that there is not enough evidence to reject the null hypothesis. The p-values for the alternative hypothesis that the time and space complexities of the best iteration's solution is on a lower order than that of the first are 0.02 and 0.05 respectively, suggesting evidence in support of the alternative hypothesis.

3) Trends in Improvement: It is evident that the results differ when considering improvement in the chosen metrics between the first and last iterations' solutions, and the first and best's. This indicated that for a sizable proportion of problems, the last solution is not the best, i.e. the improvement is not monotonic. To further investigate this notion and gather information on the trends of improvement, the proportion of problems where the first solution is best, last solution is best and neither first nor last solution is best was analyzed (see Table V).

It is observed that in a majority of cases (more than 50% for all metrics), the first solution's metric values are identical to the best solution's. Also, in a majority of cases (more than 50% for all metrics), the last solution's metric values coincide with the best solution's. There is only a small percentage of problems (less than 20% for all metrics) where neither the first nor last solution's metrics are the best across iterations.

V. IMPLICATIONS

It is not improbable that AI-powered coding assistance will thrive in the coming years. But to reach a state where AI-generated solutions can be trusted, further research and evaluation of outcomes is needed. The fact that most of the failed solutions are due to logical errors, stresses that software engineers would still have to intervene to detect and debug the problem. The key takeaway is that human experience will play a paramount role in the dawn of this new era.

At least for common and recurring programming tasks, developers may be relieved and thus focus on validating solutions and fixing errors in the generated code. And it seems that this is the direction, at least for the moment, that GitHub is trying to achieve with the new Copilot X feature that is rolling out[9]. Copilot X is going to be the technological successor of Copilot and is specifically described as a helping tool. This automation could shift the developer's focus from mundane to more complex and creative aspects of software development. Such a shift holds the promise of efficiency gains and innovation acceleration.

However, this potential benefit is contingent upon the reliability and trustworthiness of the AI-generated code. We note that no generated code contained syntax errors, i.e., all generated programs (across both models) are runnable. Thus, we stress the need for rigorous testing and validation, which should encompass not only correctness but also other runtime quality attributes such as performance. Without it, teams may find themselves at a higher risk of deploying subpar (or plain incorrect) solutions or even having to spend more time (than nowadays) on code review. So it is clear, at least for the time being, that the blind usage of these tools could result in more errors and require more time in the error resolving phase.

Another observed limitation is that ChatGPT was able to rectify only a minority of problems with initially incorrect solutions, indicating that it is not very effective at using feedback around errors to debug snippets of code. However, the average number of iterations it took to correct the ones that were eventually fixed was half of the maximum number of iterations performed, and it is possible that the correction rate could have been higher if the iteration limit had been increased. Ultimately, the interactive nature of models like ChatGPT offers promising avenues for iterative improvement of code. But, for that, more sophisticated feedback mechanisms need to be developed.

Moreover, since there are several cases where both the first and last solution coincide with the best (the proportion is well above 80% in both time and space complexity, for instance), it indicates that either there is an increase in metric values in the iterations in between or that the values remain almost constant throughout. Considering that the cases where the first solution is best and where the last solution is best do not necessarily coincide, there can be both a general increase and a general decrease in metric values across iterations. In the small fraction of cases where neither the first nor the last solution is the best, there is conclusively an initial improvement followed by a decline in each metric. This raises the question of ChatGPT's ability to not only improve upon a previous solution but to identify the changes as an improvement or deterioration.

Machine-learning models are trained on existing datasets, and an LLM, in the simplest terms, is a black box that solves the problem of predicting the next word in a sequence. Thus, whether or not the attention mechanisms used in LLMs are capable of identifying 'positive changes' (i.e., predicting the next token that constitutes an improvement) is an open question. The continuous improvement and enlargement of LLMs may help in this regard, but it is also possible that the problem is more fundamental and requires an adjustment to an LLM's architecture particularly tailored to the task of code generation. In either case, the ability to identify positive changes seems to play a crucial aspect in the iterative improvement of code, and further research is needed to determine how to enable LLMs to do so.

VI. Threats to Validity

In this section, we present the construct, reliability, and external threats of our study according to Runeson et al. [14].

Construct validity refers to the extent to which the phenomenon under study represents what is investigated according to the research questions. To mitigate this threat, we established a research protocol to guide the case study, which was thoroughly reviewed by three experienced researchers in the domain of empirical studies. Still, we acknowledge the challenges in ensuring that our selected metrics accurately represent the intended constructs. Our effort to mitigate these threats involves a careful selection of metrics that align with standard practices in software engineering research, ensuring that they are objective in the context of evaluating source.

External validity deals with possible threats when generalizing the findings derived from the sample to a broader population. Firstly, in our study we focused on ChatGPT (version 3.5) and Copilot, so our results can not be generalized for all AI coding assistants that exist, or for newer versions of GhatGPT and Copilot. Secondly, we used 60 problems from LeetCode in order to test our hypothesis, but there are a total of 1800 problems in this platform. We acknowledge again that while LeetCode problems are designed primarily to test algorithmic thinking and coding proficiency, they often lack the complexity and multifaceted nature of real-world software projects. That said, they do provide a valuable framework for honing fundamental programming skills and algorithmic efficiency, which are essential in software development. Finally, we cannot generalize our results to other programming languages, since we only tested Pyhton, and we expect that a difference is possible.

Lastly, *reliability* of the study is related to whether the data are collected and analyzed in a way that can be replicated. To minimize potential reliability threats during the data collection process, we followed specific steps for each problem and we recorded all the responses. Also, to assure correct data collection and analysis, three researchers from different institutions collaborated. Finally, since the data collection and analysis of the ChatGPT solutions involves multiple steps, we also provide supplementary material[10] with collected and processed data and results. We note that the material has been anonymized for the review.

[9] https://github.com/features/preview/copilot-x

[10] https://anonymous.4open.science/r/llm-leetcode-data-FF32/

VII. Conclusion

Prompting the question "*Would you trust a program generated by ChatGPT?*" to itself, the generated response[11] points to the fact that "*Machine learning algorithms can be very effective at solving certain problems, but they also have limitations and can produce errors if not properly designed and trained.*" It then goes on with highlighting that it is important to "*thoroughly test any program and assess its performance and accuracy before trusting it*" and that it is also important to "*consider the potential biases and ethical implications*."

This study joins the efforts for critically assessing the contribution of AI in developing software. In particular, we first examined GitHub's Copilot and OpenAI's ChatGPT ability to solve 60 random problems from LeetCode, and found that the latter provided approx. 15% more correct solutions and that the main cause of errors in both cases is incorrect logic. Next, given ChatGPT interactive paradigm, we sought to fix incorrect solutions by proving LeetCode's feedback and improve the overall code quality. The results showed that the improvement in solutions over iterations was not consistently linear, indicating variability in solution quality. The study, limited to Python, suggests the need for further research across different programming languages and more complex coding challenges, which may also help unlock the power explainable AI for source code generation.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010.

[2] J. Harper, "Interview insight: How to get the job," in *A Software Engineer's Guide to Seniority*. Apress, 2022, pp. 19–28.

[3] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of GitHub Copilot's code generation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. Singapore, Singapore: ACM, 2022, pp. 62–71.

[4] 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 *Proceedings of the Genetic and Evolutionary Computation Conference*. Boston, MA, USA: ACM, 2022, pp. 1019–1027.

[5] S. Imai, "Is GitHub Copilot a substitute for human pair-programming? an empirical study," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. Pittsburgh, PA, USA: ACM, 2022, pp. 319–321.

[6] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New Orleans, LA, USA: ACM, 2022, pp. 1–7.

[7] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586030

[8] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001292

[9] 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 *Proceedings of the 2022 IEEE Symposium on Security and Privacy*. IEEE, 2022, pp. 754–768.

[10] G. L. Scoccia, "Exploring early adopters' perceptions of chatgpt as a code generation tool," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2023, pp. 88–93.

[11] A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, M. S. Aktar, and T. Mikkonen, "Towards human-bot collaborative software architecting with chatgpt," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 279–285. [Online]. Available: https://doi.org/10.1145/3593434.3593468

[12] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, ser. SAST '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 15–24. [Online]. Available: https://doi.org/10.1145/3624032.3624035

[13] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, 2022, pp. 1–5.

[14] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[15] T. Yin, "A simple code complexity analyser without caring about the c/c++ header files or java imports, supports most of the popular languages," *from https://github.com/terryyin/lizard*, pp. 21–7, 2020.

[16] A. Kumar, "Measure code quality using cyclomatic complexity - data analytics," available at https://vitalflux.com/cyclomatic-complexity-used-measure-code-quality/#How_could_Cyclomatic_Complexity_help_measure_or_rather_predict_Code_Quality, 2022.

[17] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[11] Generated by https://chat.openai.com/chat on April 5th, 2023