Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues

YUE LIU, Monash University, Australia
THANH LE-CONG, The University of Melbourne, Australia
RATNADIRA WIDYASARI, Singapore Management University, Singapore
CHAKKRIT TANTITHAMTHAVORN, Monash University, Australia
LI LI, Beihang University, China
XUAN-BACH D. LE, The University of Melbourne, Australia
DAVID LO, Singapore Management University, Singapore

Since its introduction in November 2022, ChatGPT has rapidly gained popularity due to its remarkable ability in language understanding and human-like responses. ChatGPT, based on GPT-3.5 architecture, has shown great promise for revolutionizing various research fields, including code generation. However, the reliability and quality of code generated by ChatGPT remain unexplored, raising concerns about potential risks associated with the widespread use of ChatGPT-driven code generation.

In this paper, we systematically study the quality of 4,066 ChatGPT-generated code implemented in two popular programming languages, i.e., Java and Python, for 2,033 programming tasks. The goal of this work is three folds. First, we analyze the correctness of ChatGPT on code generation tasks and uncover the factors that influence its effectiveness, including task difficulty, programming language, time that tasks are introduced, and program size. Second, we identify and characterize potential issues with the quality of ChatGPT-generated code. Last, we provide insights into how these issues can be mitigated. Experiments highlight that out of 4,066 programs generated by ChatGPT, 2,757 programs are deemed correct, 1,081 programs provide wrong outputs, and 177 programs contain compilation or runtime errors. Additionally, we further analyze other characteristics of the generated code through static analysis tools, such as code style and maintainability, and find that 1,933 ChatGPT-generated code snippets suffer from maintainability issues. Subsequently, we investigate ChatGPT's self-debugging ability and its interaction with static analysis tools to fix the errors uncovered in the previous step. Experiments suggest that ChatGPT can partially address these challenges, improving code quality by more than 20%, but there are still limitations and opportunities for improvement. Overall, our study provides valuable insights into the current limitations of ChatGPT and offers a roadmap for future research and development efforts to enhance the code generation capabilities of AI models like ChatGPT.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software creation and management**.

Additional Key Words and Phrases: Automated code generation, ChatGPT, code analysis

Authors' addresses: Yue Liu, yue.liu1@monash.edu, Monash University, Australia; Thanh Le-Cong, congthanh.le@student.unimelb.edu.au, The University of Melbourne, Australia; Ratnadira Widyasari, ratnadiraw.2020@phdcs.smu.edu.sg, Singapore Management University, Singapore; Chakkrit Tantithamthavorn, chakkrit@monash.edu, Monash University, Australia; Li Li, lilicoding@ieee.org, Beihang University, China; Xuan-Bach D. Le, bach.le@unimelb.edu.au, The University of Melbourne, Australia; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore.

XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. 1, 1 (July 2023), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since launching in November 2022, ChatGPT [29], an AI-powered chatbot developed by OpenAI, has rapidly gained popularity. Within just two months, ChatGPT had reached 100 million unique users, surpassing even the fastest-growing social network, TikTok, in user acquisition [41]. Due to its remarkable ability in language understanding and human-like answering, ChatGPT has shown great promise in revolutionizing various research fields, including code generation due to it being trained on extensive repositories of source code [29]. Interestingly, users without any coding experience can use the model to generate code snippets from natural language requirements. Although ChatGPT's ability on code generation tasks has been *informally* receiving positive feedback from the community, there exists no study that *formally* investigates the reliability and quality of code generated by ChatGPT.

Despite the great promise of ChatGPT in code generation, formally and thoroughly studying the reliability and quality of code generated by ChatGPT is becoming increasingly critical. This is due to ChatGPT now being used not only by professional developers but also by novice programmers and individuals with no coding experience. Code quality issues in ChatGPT-generated code, if not properly identified and addressed, may unduly affect code comprehension, introduce bugs, or create security vulnerabilities in users' projects. Consequently, the widespread adoption of ChatGPT for code generation could potentially lead to a decline in the overall quality of software systems. Therefore, it is crucial to examine and address the common code quality issues that may arise from using ChatGPT-generated code.

In this paper, motivated by the above challenges, we are the first to formally study the reliability and quality of ChatGPT-generated code. Our objectives are (1) to analyze the correctness of ChatGPT-generated code, (2) to identify and characterize code quality issues that may arise, and (3) to examine different prompts that leverage feedback from static analysis tools and runtime errors to guide ChatGPT in mitigating code quality issues. Through experiments addressing the following three research questions, our work provides valuable insights that help increase awareness within the community regarding code quality issues in ChatGPT-driven code generation.

- **RQ1:** ***(Performance)*** *How effective is ChatGPT on code generation for programming tasks?*
- **RQ2:** ***(Bugs and Issues)*** *What are the common issues in ChatGPT-generated code?*
- **RQ3:** ***(Repair with Prompting)*** *Can ChatGPT fix the code quality issues with prompting?*

To answer these questions, we first construct a benchmark dataset containing a total of 2,033 programming tasks from LeetCode, with 501 classified as easy, 1,064 as medium, and 468 as hard. We then evaluate the ChatGPT-generated code for these programming tasks against LeetCode's test suite to evaluate ChatGPT's performance on code generation. Next, we employ static analysis tools including Pylint [42], Flake8 [10], PMD [9], and CheckStyle [4] to examine ChatGPT-generated code. Based on feedback from static analysis tools and runtime errors, we conduct an open card sort discussion [38] to characterize common code quality issues including compilation and runtime errors, wrong outputs, code style and maintainability, and performance and efficiency. Finally, we attempt to mitigate the identified code quality issues by using several fixing-prompts, i.e., prompts that request ChatGPT to fix issues. To do so, we experiment with fixing-prompts with and without feedback from static analysis tools and runtime errors.

Our experimental results lead to the following findings: (1). On various code generation tasks on Python and Java, 66% and 69% of Python and Java programs generated by ChatGPT are functionally-correct programs, i.e., programs that pass all given test cases. We observed that the performance is attributed to various factors such as task difficulty, the time when tasks are introduced, and program size. Especially, ChatGPT's performance drops up to five times on new programming tasks introduced after January 2022, highlighting the model's limitations in adapting to new programming tasks. (2). We also identified that the generated code commonly suffers from different code quality issues, such as compilation and runtime errors, wrong outputs, code style and maintainability issues. For instance, among ChatGPT-generated code that passed the test cases, 53% of the Java code and 37% of the Python code exhibited code style and maintainability issues. This highlights the importance of addressing such problems to ensure the long-term success of AI-driven code generation. In other words, developers and users still need to take appropriate measures to improve the overall quality of the ChatGPT-generated code. (3). Our study on ChatGPT's self-debugging capabilities revealed that ChatGPT can partially fix code quality issues in the generated code with feedback from static analysis tools and runtime errors. Moreover, the effectiveness of ChatGPT in addressing code quality issues varies depending on the feedback information, programming languages, and code quality issues.

To the best of our knowledge, our work is the first to:

- Conduct a comprehensive study to evaluate the reliability and quality of ChatGPT-generated code;
- Identify and characterize common code quality issues in ChatGPT-generated code;
- Introduce a new dataset comprising 2033 programming tasks and 4066 ChatGPT-generated code snippets implemented in two popular programming languages: Java and Python, with 2,556 code with quality issues;
- Conduct an exploration study on ChatGPT's self-debugging capability for code quality issues.

To support the open science initiative, we publish the studies dataset and a replication package, which is publicly available at https://doi.org/10.6084/m9.figshare.22773665

2 BACKGROUND

2.1 Large Language Model

Large Language Models (LLMs) have achieved impressive performance on a wide range of natural language processing (NLP) tasks, including machine translation [8, 17, 34], summarization [12, 15, 34], sentiment analysis [48, 49], and question-answering [31, 34]. These models, typically based on deep learning architectures such as transformers, are trained on massive amounts of text data, allowing them to learn complex language patterns and structures. By capturing both the syntax and semantics of human language, LLMs have been successful in generating coherent and contextually relevant text.

One prominent example of an LLM is ChatGPT, developed by OpenAI and based on the GPT-3 architecture. ChatGPT demonstrates an unprecedented ability to understand and generate human-like text, making it well-suited for a variety of applications, including code generation. By training ChatGPT on extensive source code repositories, the model has become capable of generating code snippets and solving programming problems with remarkable accuracy.

2.2 Motivation

While LLMs have shown great promise in code generation, the reliability of the generated code is questionable. The problem has become more critical with the emergence of ChatGPT, as LLMs-driven code generation is now being used not only by experienced developers but also by novice

programmers or even individuals with no coding experience, who may be unaware of the code quality issues.

Code 1. A simplified version of an buggy code generated by ChatGPT for solving the LeetCode Problem 1093 - 'Statistics from a Large Sample'

```
1 #Description
2 '''
3 You are given a large sample of integers in the range [0, 255]. Since the sample
      is so large, it is represented by an array count where count[k] is the number
      of times that k appears in the sample.
4 Calculate `mean`, i.e. the average of the sample, calculated as the total sum of
      all elements divided by the total number of elements.
5 Input: count = [0,1,3,4, 0, ..., 0]
6 Output: 2.37500
7 Explanation: The sample represented by count is [1,2,2,2,3,3,3,3] so the mean is
8     1+2+2+2+3+3+3+3)/8 = 19/8 = 2.375.
9 '''
10 #Chat GPT Generated Code
11 class Solution(object):
12    def sampleStats(self, count):
13        countSum = 0
14        totalNum = sum(count)
15        mean = 0
16        for i in range(256):
17            if count[i] > 0:
18                countSum += count[i] * i
19        mean = countSum / totalNum
20        return mean
```

Code 1 illustrates a motivating example for our study. This is a simplified version of buggy code generated by ChatGPT (lines 11 - 20) for solving the LeetCode Problem 1093 - 'Statistics from a Large Sample' (lines 3-8). The problem requires ChatGPT to generate a code that calculates the mean of a large sample of integers, represented by a `count` array where `count[k]` represents the frequency of integer k in the sample. While looking straightforward and correct, the ChatGPT-generated code produces the incorrect output from the example test given in lines 5-8. The expected output from the test is 2.375, while the result from ChatGPT-generated code is 2. The root cause of the error is line 19 in which `mean` is calculated using integer division (rounding down to integer) instead of division as both `countSum` and `totalNum` are integers. Though the error is quite simple, it can be difficult for developers or programmers who are not familiar with Python programming languages to detect. It can also lead to more complex errors in other functions that call to this function without the awareness of the error.

Code 2. A code smell generated by ChatGPT for solving the LeetCode Problem 1838 - 'Frequency of the Most Frequent Element'

```
1 def getMinDistance(self, nums: List[int], target: int, start: int) -> int:
2     min_diff = float('inf')
3     min_index = -1
4     for i in range(len(nums)):
5         if nums[i] == target:
6             diff = abs(i - start)
7             if diff < min_diff:
8                 min_diff = diff
```

```
9                 min_index = i
10    return min_diff
```

Additionally, we also observed that the quality of the ChatGPT-generated code may still be poor even if it is functionally correct. Code 2 illustrates an example of poor-quality code generated by ChatGPT. This is a simplified version of code generated by ChatGPT for LeetCode Problem 1838, 'Frequency of the Most Frequent Element'. The `min_index` variable is declared on line 3 and assigned values on line 9, but it is never used elsewhere in the code. This is a minor code smell, but it is worth noting that this issue occurs in a simple 10-line code for a common problem. Let's imagine complex tasks and code, could we ensure that ChatGPT-generated code does not contain smells, bugs, or even vulnerabilities? This realization motivated us to conduct a comprehensive study on the quality issues present in ChatGPT-generated code. Our study aims to not only enhance our understanding of these issues but also to provide suggestions for mitigating them.

3 STUDY DESIGN

In this section, we present the details of our study design. We first present our research questions and then introduce our dataset and the ChatGPT model.

3.1 Research Question

In this empirical study, we aim to answer the following research questions.

RQ1. *How effective is ChatGPT on code generation for programming tasks?* Despite informally receiving positive feedback from the community, there is a lack of comprehensive study on the performance of ChatGPT in code generation. This research question aims to measure how well ChatGPT could generate code for programming tasks and to analyze the factors that impact its performance, including task complexity, difficulty, and time that tasks are introduced.

RQ2. *What are the common issues in ChatGPT-generated code?* This research question aims to analyze issues in ChatGPT-generated code using popular static analysis tools and categorize them into common categories.

RQ3. *Can ChatGPT fix the code quality issues with prompting?* Conversational AI models such as ChatGPT allow users to provide feedback to allow ChatGPT to revise its output. This research question aims to investigate whether ChatGPT can correct coding issues based on runtime errors, feedback from the compiler, and static analysis tools.

3.2 Constructing Benchmark Dataset

Existing benchmarks for evaluating AI-based code generation are often limited and outdated. For example, prior work [5, 6] has relied on the HumanEval dataset [6], which only includes 164 Python programming tasks before 2021. Such small and outdated datasets, however, are not ideal for evaluating modern AI models, such as ChatGPT, since they lack diversity and may have been used in the training data of modern AI models. To address this issue, Fan *et al.* [11] introduce a new dataset, LMDefects, that contains 113 Java programming tasks released after Jun 2021. The dataset was collected from LeetCode, a well-known online platform that offers a variety of coding challenges to help programmers enhance their abilities and prepare for technical interviews. The dataset, however, is still relatively small and focused solely on Java programming tasks.

In this study, we extend LMDefects by collecting all accessible programming tasks and the relevant official public test suites in LeetCode, and investigate ChatGPT's ability in generating code in both Java and Python. At the time of data collection (March 2023), there were 2,617 task problems available on LeetCode. These problems cover various topics, including data structures, algorithms, databases, and concurrency. For our dataset, we focused on the problems that were

Table 1. Zero-shot pass rate accuracy on LeetCode

pass@1	Easy	Medium	Hard	Overall
Python	0.890	0.674	0.400	0.664
Java	0.860	0.710	0.468	0.691
P-value	0.346	0.654	0.535	0.471
Effect Size	0.015	-0.012	-0.005	0.002

designed specifically for Java and Python, as these two languages are widely used and have a large community of developers. Additionally, in order to provide a fair and accessible dataset, we filtered out the premium tasks that require a subscription to access. After this filtering process, we successfully collected 2,033 programming tasks from LeetCode. Out of the 2,033 tasks in our dataset, we found that 501 were classified as easy, 1,064 as medium, and 468 as hard.

3.3 The ChatGPT Model

ChatGPT is a large language model that can provide detailed responses given natural language instructions. We use the model that has been fine-tuned from the GPT3.5 models [32] using Reinforcement Learning from Human Feedback [39]. In our study, we used the ChatGPT-March-23 version [30], which was trained on data up to 2021. To instruct ChatGPT, we followed the zero-shot prompt setting which does not include examples of code generation in the prompt as follows: "*Please provide a code implementation of the following description + <description of a programming task> + Provide a valid <programming language> code with this template <solution template provided containing the input and output specifications>*".

4 RQ1: PERFORMANCE

In this section, we present the results for RQ1, which investigates the effectiveness of ChatGPT in code generation. To mitigate the randomness of ChatGPT, we make ChatGPT deterministic by setting the temperature to 0 and running the model once for each task, using the first generated output for evaluation. ChatGPT's performance is measured with zero-shot pass-rate (*pass@1*), which measures whether the model produces a correct solution on the first attempt. For example, if ChatGPT generates code snippets for 10 tasks and 7 of them pass the test cases in the first attempt, the pass@1 accuracy would be 0.70. We also conducted the Mann-Whitney U rank test [26] to measure the statistical significance of the performance differences by ChatGPT across factors. The Mann-Whitney U rank test is a non-parametric statistical test used to compare two independent samples to determine whether there is a significant difference between the two distributions, while the Cliff's Delta [25] effect size measures the magnitude of the difference between the samples.

Table 1 presents the pass rate of ChatGPT for LeetCode tasks with different difficulties in both Python and Java. It can be seen that ChatGPT performs better on easy tasks than on medium and hard tasks. For Python, the model achieves a pass@1 accuracy of 0.890 for easy tasks, indicating that ChatGPT can handle 89% of easy tasks in one attempt. However, the performance drops to 0.674 for medium tasks and further decreases to 0.400 for hard tasks. Similarly, for Java, the model attains a pass@1 accuracy of 0.860 for easy tasks, 0.710 for medium tasks, and 0.468 for hard tasks. These findings suggest that the difficulty level of tasks has a significant impact on the performance of ChatGPT in code generation. Table 1 also shows the results from the Mann-Whitney U test on performance differences between Python and Java. Although ChatGPT performs slightly better in

Table 2. Effect Sizes and P-values for Pass vs Fail Comparisons in Python and Java

Comparison (@pass *vs* @fail)	Language	P-value	Effect Size (Cliff's Delta)
Time period	Python	<0.001	0.511
	Java	<0.001	0.446
Program length	Python	<0.001	0.249
	Java	<0.001	0.309

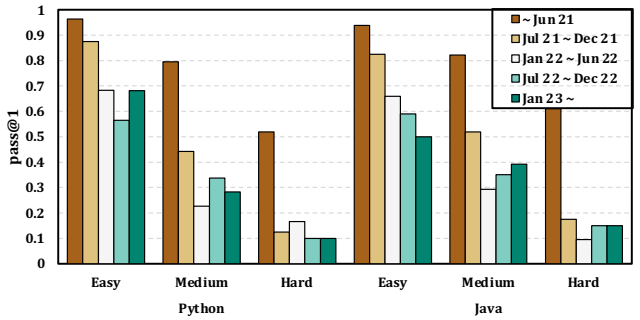

Fig. 1. Pass Rate by Difficulty and Time Period

Java for medium (↑ 5.3%) and hard tasks (↑ 17%), their difference in performance is not significant with a p-value of at least 0.53 and an effect size value less than 0.02 [25].

As ChatGPT (GPT-3.5-turbo) is trained solely on data until September 2021 [29], it's also important to measure its performance changes as new challenges arise. Figure 1 illustrates the pass rates of ChatGPT across different difficulty levels (easy, medium, and hard) and programming languages (Python and Java) over five distinct time periods. The chart shows that the performance of ChatGPT declines over time for both Python and Java. Specifically, ChatGPT can solve more than half of the hard-level code tasks before June 2021, but its performance reduces drastically to nearly 0.1 for the subsequent time periods. The decline in performance is not as pronounced for easy-level tasks, which indicates that ChatGPT still maintains some level of proficiency when dealing with simpler problems, even as time progresses. As shown in Table 2, the Mann-Whitney U test indicates that the time period when tasks are introduced has a statistically significant difference between passed code and failed code (p-value < 0.001) with a large Cliff's delta effect size. However, this observation also highlights the model's limitations in adapting to the intricacies and nuances of more complex, newer programming challenges. Moreover, the drop in performance of ChatGPT could be explained by a data leakage issue in which the LeetCode problem may be contained in ChatGPT's training data. Therefore, the performance of ChatGPT on old programming tasks which is published before December 2021 may only reflect the memorization capability [28] of ChatGPT instead of its real performance. Therefore, the results also highlight the need to evaluate the model on the newly-introduced dataset after September 2021 for fair evaluations.

In addition to difficulty levels and time periods, another factor that may impact the performance of ChatGPT is the length of the generated code. Figure 2 presents the pass rates of ChatGPT for both Python and Java programming languages, grouped by the number of lines in the generated code. It is worth noting that the distribution of code lengths is not uniform, with the majority of generated code snippets falling into the 10-20 lines range for Python and the 20-30 lines range for Java. This discrepancy highlights the differences in verbosity and structure between the two

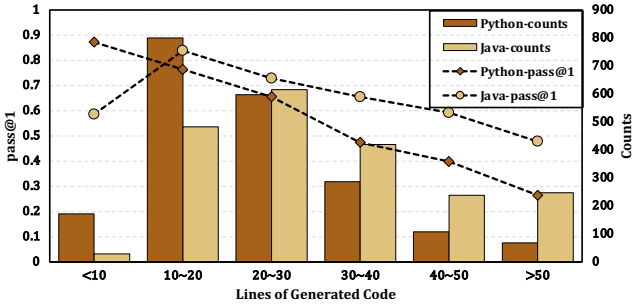

Fig. 2. Pass Rate by Length of Generated Program

programming languages, which might also contribute to the variations in ChatGPT's performance across different length categories. In Figure 2 there is a clear trend of decreasing the pass rate for both Python and Java, as the length of the generated code increases. For Python, the pass@1 rate starts at 0.872 for code snippets with less than 10 lines and gradually decreases to 0.265 for code snippets with more than 50 lines. For Java, the pass@1 rate gradually decreases from 0.838 for code snippets with 10–20 lines to 0.478 for code snippets with more than 50 lines. This trend suggests that ChatGPT's ability to generate correct and bug-free code is inversely proportional to the size of the generated code. This could be due to the increased complexity, and the greater number of potential interactions between code elements as the code size grows, making it harder for the model to generate a correct and complete solution. As shown in Table 2, the Mann-Whitney U test confirms the significance of the differences (p-value < 0.01) with a small to medium effect size. Overall, these findings suggest that improving the model's ability to generate longer and more complex code snippets is a valuable direction for future research and development.

In summary, our results indicate that the model's performance declines over the difficulty level and time period of code tasks. Furthermore, the model's ability to generate correct and bug-free code is inversely proportional to the size of the generated code, suggesting that the increased complexity of longer code snippets poses a significant challenge for the model. Based on these findings, it is recommended that future research and development efforts focus on improving the model's ability to handle more complex tasks, adapt to new programming challenges, and generate longer and more intricate code snippets.

> ***Finding 1:*** *The performance of ChatGPT is significantly and substantially affected by task difficulty, time that tasks are introduced, program size and programming languages.*

5 RQ2: BUGS AND ISSUES

5.1 Static Analysis

To address RQ2, our first step is to gather output from LeetCode for ChatGPT-generated code. If the generated code does not pass the tests, we label it as "Wrong Outputs" to indicate failure to meet the problem requirements. However, passing test cases alone does not guarantee that the code is free from quality issues. Therefore, to further investigate the code quality and identify potential bugs, style issues, and other concerns that might impact the overall quality, we employ static analysis tools tailored for each programming language.

For Java code samples, we use PMD [9] and Checkstyle [4]. PMD is a well-known static analysis tool that inspects Java source code to identify potential problems and provides suggestions for improvements [47]. Checkstyle, on the other hand, statically checks Java code against a specified

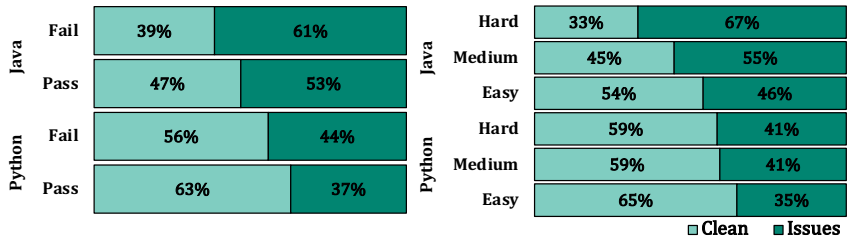

Fig. 3. Code Quality Distribution by Difficulty and Language for Passed and Failed Tasks

set of coding conventions. Both tools evaluate the Java source code against a set of rules, reporting warnings for any violations, their priority, and the corresponding lines in the file. Similarly, for Python code samples, we utilize Pylint [42] and flake8 [10]. Pylint is a popular and comprehensive static analysis tool that enforces coding standards and detects various types of issues in Python code [36, 44]. Flake8 is another widely-used tool for Python, which combines PyFlakes, pycodestyle, and McCabe to check for syntax errors, style issues, and code complexity, respectively. These tools enable us to assess the code quality from multiple dimensions, beyond just functional correctness.

After running the static analysis tools, we gather issues found for each ChatGPT-generated program identified by the compilers and the static analysis tools. In order to simplify the analysis and reduce the impact of false positives, we focus on more significant aspects of code quality and functionality. Therefore, we choose to ignore messages related to style issues such as whitespace, newline, and invalid naming conventions, which is consistent with the approach taken in prior work [36]. Figure 3 presents the distribution of code quality based on the difficulty levels and programming languages for both passed and failed tasks. The figure highlights the proportion of clean code, which refers to the code snippets without issues identified by the static analysis tools, and the code with issues.

Figure 3 shows that the proportion of clean code is generally higher for passed tasks compared to failed tasks. For Python, 63% of the passed tasks have clean code, while only 56% of the failed tasks are clean. In the case of Java, 47% of the passed tasks have clean code, as opposed to 39% for failed tasks. Additionally, it is evident that the percentage of clean code decreases as the difficulty level increases for both Python and Java. For example, the percentage of clean Java code decreases from 54% for easy tasks to 45% for medium tasks, and further drops to 33% for hard tasks. These findings underscore the importance of addressing code quality concerns in tandem with functional correctness to better support developers in handling complex programming tasks across different languages and domains.

> ***Finding 2:*** *Code quality issues commonly happen in both code that pass or failed test cases, highlighting the need for characterizing and addressing these concerns alongside functional correctness.*

5.2 Open Card Sorting Discussion

To gain a deeper understanding of the common issues and patterns found in the ChatGPT-generated code, we conducted a qualitative analysis using open card sorting. The open card sorting process has been used in many previous studies [3, 24, 45] to generate categories or taxonomy from data. In this study, we follow the card sorting process highlighted by previous studies [24, 38], which mainly consisted of two phases. First, the preparation phase where we created cards for each programming task. This card is filled with the title of the programming task, generated code by ChatGPT, the test results, and the static analysis tools result. The second phase is the execution phase where

we choose the representative random sample of 154 programming tasks for both Java and Python from a total of 2,033 programming tasks (with a 99% confidence level and 10% margin of error). We analyzed and discussed each card, and iteratively sort them into groups based on their issues. During the card sorting process, we found that many of the cards could be placed into several different categories, as there can be more than one issue coming up in a given code. Once all the cards were placed into categories, we created category names based on the issue patterns that we observed.

After a thorough analysis and discussion of the card sorting process, the experts identify four different categories of issues in the ChatGPT-generated code:

Compilation and Runtime Errors: Compilation and Runtime errors encompass issues that prevent the correct execution of the code. These errors can arise due to various factors, such as incorrect use of a programming language's syntax, improper handling of data structures, invalid input, or exceeding the bounds of an array. The errors often lead to failures during compilation or runtime, and they need to be resolved before the program can function as intended. Code 3 demonstrates a compilation error that occurs when ChatGPT attempts to use the ^(bitwise XOR) operator with incompatible operand types, causing a compilation error.

Code 3. An example of compliation error (LeetCode Problem 2564 - Java)

```java
if (prefix[mid] ^ (left == 0 ? 0 : prefix[left - 1]) > queries[i][1]) {
    r = mid - 1;
} else {
    l = mid;
}
//Compiler: Solution.java:1: error: bad operand types for binary operator '^'
```

Wrong Outputs: Wrong Outputs represent issues in the code that cause it to produce incorrect results or fail to meet the problem requirements. These errors can stem from incorrect algorithms, improper handling of edge cases, or other inaccuracies in the desired logic. These errors can occur even when the code is syntactically correct and free from any runtime errors. Code 4 presents an example where ChatGPT provided an inaccurate solution to LeetCode Problem 746, "Min Cost Climbing Stairs". The issue arises due to the incorrect construction of the loop and final return statement. In this specific example, the input is [10, 15, 20]. The expected output is 15, achieved by climbing the steps with costs of 10 and 20, while skipping the step with a cost of 15. However, the ChatGPT-generated code produces an output of 25. This error occurs because the loop iterates one step more than necessary, causing the last step's cost to be included in the calculation even when it should not be.

Code 4. A example of Wrong Outputs (LeetCode Problem 746 - Python)

```python
def minCostClimbingStairs(self, cost: List[int]) -> int:
    n = len(cost)
    dp = [0] * (n+1)
    dp[0] = cost[0]
    dp[1] = cost[1]
    for i in range(2, n+1):
        dp[i] = min(dp[i-1], dp[i-2]) + cost[i-1]
    return min(dp[n-1], dp[n])
#Compiler: AssertionError: Input: cost = [10, _15_ ,20]  Expected output: 15
```

Code Style and Maintainability: This category encompasses issues related to the code's formatting, organization, and adherence to established style guides. Poorly styled or organized code is difficult to read, understand, and maintain. Examples include inconsistent indentation, overly

long lines, or lack of meaningful variable names. Code 5 presents an example where ChatGPT provided an smelly solution to LeetCode Problem 1297, “Maximum Number of Occurrences of a Substring". In this case, the static analysis tool PMD identified an issue on line 1: the variable ‘maxSize‘ is declared but never used.

Code 5. A example of Code Style and Maintainability (LeetCode Problem 1297 - Python)

```
def maxFreq(self, s: str, maxLetters: int, minSize: int, maxSize: int) -> int:
    count = defaultdict(int)
    res = 0

    for i in range(len(s) - minSize + 1):
        substring = s[i: i + minSize]
        if len(set(substring)) <= maxLetters:
            count[substring] += 1
            res = max(res, count[substring])

    return res
#PMD: 1297-maximum-number-of-occurrences-of-a-substring.py:1:61: W0613: Unused
    argument 'maxSize' (unused-argument)
```

Performance and Efficiency: Performance and efficiency issues arise when the code is not optimized for execution speed or resource usage. These issues can lead to slow or resource-intensive programs, which can negatively impact the user experience or system stability. Examples include inefficient algorithms, unnecessary memory allocations, or redundant calculations. For example, when using ChatGPT to solve LeetCode Problem 1982, titled "Find Array Given Subset Sums", the compiler outputs a “TIMEOUT" error due to inefficient loop control within the generated code.

> ***Finding 3:*** *Issues in ChatGPT-generated code can be categorized into four categories: Compilation & Runtime Errors, Wrong Outputs, Code Style & Maintainability, Performance & Efficiency.*

5.3 Quantitative Analysis

In order to gain a comprehensive understanding of the issues present in the ChatGPT-generated code, we perform a quantitative analysis on the categorized issues identified in the open card sorting discussion. This analysis aims to provide insights into the frequency, distribution, and nature of the issues across different difficulty levels and programming languages. From the card sorting results, we derive rules to classify the issues in the generated code, allowing us to identify areas where ChatGPT performs well and aspects that require improvement. It is important to note that one generated code snippet may contain multiple issues, which can further affect the analysis. By highlighting these issues, our analysis can guide future research and development efforts to enhance the code generation capabilities of ChatGPT and similar AI models.

5.3.1 Overall Analysis. Table 3 presents the distribution of the four issues across task difficulty levels and programming languages. From the table, it is evident that Compilation & Runtime Errors and Performance & Efficiency issues are relatively less frequent, indicating that ChatGPT is generally successful in generating syntactically correct and efficient code. However, Wrong Output and Code Style & Maintainability issues are more prevalent and tend to be the most common challenges faced by the generated code. Specifically, 1,081 out of 4,066 generated code snippets (i.e., 26.6%) exhibit wrong output, while 1,933 out of 4,066 (i.e., 47.5%) encounter issues related to code style and maintainability. Furthermore, as the difficulty level of the tasks increases, the prevalence of these issues also tends to rise. For example, 7 out of 501 generated code snippets (i.e., 1.4%) for

Table 3. Distribution of issues across difficulty levels and programming languages. P and J denotes Python and Java, respectively.

	Easy N=501		Medium N=106		Hard N=468		
	P	**J**	**P**	**J**	**P**	**J**	**Sum**
Compilation and Runtime Error	7	8	37	32	46	47	177
Wrong Output	47	60	290	259	229	196	1081
Code Style and Maintainability	176	230	432	588	194	313	1933
Performance and Efficiency	1	2	20	16	6	6	51

Table 4. Comparison of Common Compilation and Runtime Error Categories in Java and Python Programs

Category	Description	Java Count	Python Count
Division by Zero	Attempt to divide by zero	3	3
Illegal Index	Accessing an array or list with an invalid index	45	25
Concurrent Modification	Modifying a collection during iteration	1	1
Empty Collection Access	Accessing an element from an empty collection	2	3
Key Not Found	Accessing a non-existent key in a dictionary or map	1	13
Null Reference	Attempt to access an attribute or method of a null object	8	4
Type Mismatch	Using an incorrect data type in an operation or function call	6	27
Resource Limit Exceeded	Exceeding the system's resource limits	2	1
Syntax error	Incorrect syntax or structure in the code	4	0
Undefined Variable	Accessing or using a variable that has not been defined	8	6
Attribute Not Found	Attempt to access a non-existent attribute or method of an object	3	7
Duplicate Variable	Defining a variable more than once in the same scope	4	0

easy Python tasks exhibit compilation and runtime errors, while the number of execution errors increases to 46 out of 468 (i.e., 9.8%) for hard Python tasks. These findings indicate that ChatGPT, while powerful, has room for improvement in automated code generation to deliver more reliable and effective AI-generated code.

> ***Finding 4:*** *Wrong Outputs and Code Style & Maintainability issues are the most common challenges faced by the ChatGPT-generated code while Compilation & Runtime Errors and Performance & Efficiency issues are relatively less prevalent.*

5.3.2 Analysis on Compilation & Runtime Errors. Table 4 presents a comparison of common compilation and runtime errors error categories in Java and Python programs (i.e., 80 Python and 97 Java programs with the errors). From this table, we can observe that ChatGPT generates code containing a diverse range of errors across multiple categories, indicating the need for improvement in various aspects of code generation. Additionally, a significant portion of common compilation and runtime errors are relevant to the semantics of the generated program. For example, these errors may contain illegal values (e.g., division by zero or invalid indices) and wrong access (e.g., concurrent modification, null references, and empty collection access). These observations can be explained by the probabilistic nature of the ChatGPT model, which predicts subsequent tokens based on preceding ones. This nature enables ChatGPT to understand the semantics of common programs that appear frequently in the training set. However, the model captures the semantics implicitly from the training data, leading to misunderstandings of program semantics and subsequently resulting in semantically-related compilation and runtime errors. These findings indicate that

Table 5. Top 10 Issues Affecting Code Style and Maintainability in Python Programs Generated by ChatGPT

Errors	Descriptions	Pylint	Flake8	#Programs
ConsiderUsingEnumerate	Used when code that iterates with range and len is encountered.	x		213
NoElseReturn	Used in order to highlight an unnecessary block of code following an if containing a return statement.	x		161
UnusedVariable	Used when a variable is defined but might not be used.	x	x	103
RedefinedBuiltin	Used when a variable or function override a built-in.	x		63
ConsiderUsingDictItems	Used when iterating over the keys of a dictionary and accessing the value by index lookup.	x		39
AvoidAmbigousNames	Used when code use variables named 'I', 'O', or 'l'		x	38
TooManyBranches	Used when a function or method has too many branches, making it hard to follow.	x		36
TooManyLocals	Used when a function or method has too many local variables.	x		32
BlankLines	Nested functions should contain 1 blank line between their definitions.		x	28
InconsistentReturnStatements	Either all return statements in a function should return an expression, or none of them should.	x		27

Table 6. Top 10 Issues Affecting Code Style and Maintainability in Java Programs Generated by ChatGPT

Errors	Descriptions	CheckSyle	PMD	#Programs
MultipleVariableDeclarations	Each variable declaration must be in its own statement	x		334
AvoidReassigningParameters	Emitted when incoming parameters are reassigned values		x	176
ForLoopCanBeForeach	Used to recommend to use foreach loop instead of loop.		x	114
RedundantModifier	Emitted when a modifier is redundant	x		112
RightCurly	Emitted when right curly in a code violate common conventions	x		87
VisibilityModifier	Used to recommend that a variable should not public	x		86
NPathComplexity	Used when a method have too many acyclic execution paths		x	81
LooseCoupling	Used when using implementation types instead of interface		x	64
HiddenField	Emitted when a local variable or a parameter does not shadow a field that is defined in the same class.	x		55
UseConcurrentHashMap	Recommend to use the ConcurrentHashMap implementation		x	54

incorporating semantic information into ChatGPT could potentially improve the quality of the generated code, indicating a promising direction for future research.

> ***Finding 5:*** *ChatGPT-generated code contains various types of execution errors, primarily due to misunderstandings of program semantics.*

We also notice that Illegal Index errors are quite prevalent in both languages, particularly in Java. In fact, out of the 97 compilation and runtime errors encountered in Java, 45 of them (46.4%) are attributed to using an invalid index. Meanwhile, Type Mismatch errors are more prevalent in Python than in Java, with 27 occurrences in Python compared to 6 in Java. This observation could be due to Python's dynamic typing system, which allows for more flexibility in variable types, but can also lead to unexpected type-related issues at runtime. Overall, these findings suggest that different languages may have distinct compilation and runtime error patterns and that improvements in code generation should take these language-specific characteristics into account. Additionally, the presence of various errors highlights the need for more effective debugging and error detection tools tailored to each language, ultimately leading to more robust and efficient code generation.

> ***Finding 6:*** *Different languages may have distinct compilation and runtime error patterns.*

5.3.3 Analysis on Code Style & Maintainability. Tables 5 and 6 present the top 10 issues affecting code style and maintainability in Python and Java programs generated by ChatGPT, respectively. From these tables, we can see various types of code styles and maintainability issues in the ChatGPT-generated code.

In Python, the top three issues are ConsiderUsingEnumerate (213 out of 2033 programs, 10.5%), NoElseReturn (161 out of 2033 programs, 7.9%), and UnusedVariable (103 out of 2033 programs, 5.1%). Interestingly, 5.1% of ChatGPT-generated code has unused variables, which is considered a bad smell in code quality. Meanwhile, MultipleVariableDeclarations, AvoidingReassigningParameters, ForLoopCanBeForEach, and RedundantModifier are the most frequent issues happening in Java code

generated by ChatGPT, accounting for more than 36% ((334+176+114+112)/2033) of the generated code. The presence of these issues indicates that the code quality of ChatGPT-generated code for both Python and Java is not perfect and could be improved.

We further compare code style and maintainability issues in Java and Python. Our results show that there are no overlapping top-10 issues in Python and Java. The possible reason is that Python and Java have very different code styles and common practices. These results highlight the need for language-specific techniques to address the issues. Finally, by analyzing the issues detected by different static analysis tools, we can see that there is only one common issue in Python that can be detected by both Pylint and Flake8. Similarly, there is no overlap between CheckStyle and PMD. Thus, using multiple static analysis tools can provide a more comprehensive analysis of code style and maintainability in ChatGPT-generated code.

> ***Finding 7:*** *ChatGPT-generated code contains various types of code style and maintainability issues. Their common issues are specific to the language and tool being used.*

6 RQ3: REPAIR WITH PROMPTING

Sections 4 and 5 have demonstrated that ChatGPT is capable of generating functional code for various code generation tasks. However, the generated code sometimes suffers from different code quality issues, such as execution errors, wrong outputs, and maintainability problems. Addressing these issues is vital to ensure the reliability and efficiency of the generated solutions. Unlike traditional code generation tools, ChatGPT has the potential to learn from user interactions and refine its outputs based on the feedback it receives. This interactive process can lead to more accurate and high-quality code generation. In this section, we investigate the self-debugging capabilities of ChatGPT in addressing the code quality issues identified in the generated code. We focus on providing effective feedback and exploring various strategies to enhance the performance of the model. To investigate the impact of user feedback on the code quality of ChatGPT-generated solutions, we employ two types of feedback: (1) Simple Feedback and (2) Feedback with Static Analysis.

Simple Feedback: This type of feedback involves providing ChatGPT with basic information about the issues in the generated code. For example, if a code quality issue is detected in the code, we provide feedback to ChatGPT as follows: "*The generated code has quality issues. Please provide a better code implementation as expected by the task description.*"

Feedback with Static Analysis and Runtime Errors: In this method, we utilize the insights from static analysis tools and runtime errors (as discussed in Section 5) to offer more precise and detailed feedback to ChatGPT. Thus, we augment the simple feedback with additional information derived from static analysis reports and runtime error messages. For example, if a static analysis tool pinpoints a specific error or poor coding practice, we supply ChatGPT with feedback that directly addresses the particular issue as follows: "*The generated code contains the following quality issues: + <details from static analysis tools> + Please provide a better code implementation as expected by the task description.*"

We use both types of feedback to prompt ChatGPT to refine and improve its generated code. Then, we compare the revised code with the original version to evaluate the effectiveness of the feedback in addressing the identified code quality issues.

Figure 4 presents the fixed rates for different feedback types and code quality issues for both Java and Python. The fixed rate is defined as the proportion of code quality issues that were successfully addressed and resolved by ChatGPT if the issue no longer happens, measured as a percentage (i.e., Fix Rate = $\frac{\text{Number of Issues Resolved}}{\text{Total Number of Issues}}$). Overall, Figure 4 presents that ChatGPT can successfully repair

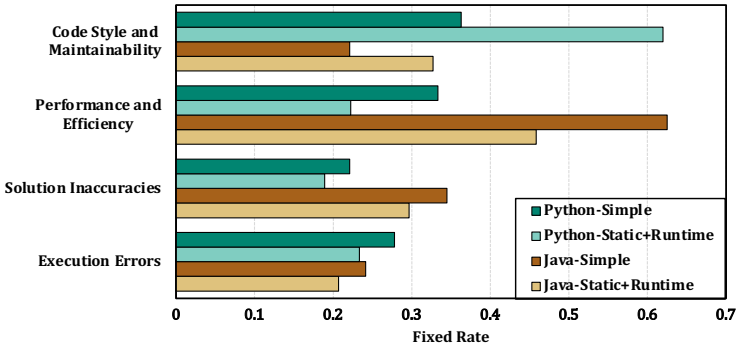

Fig. 4. Comparison of Fix Rates for Different Feedback Types and Code Quality Issues

from about 20% to 60% code quality issues itself. Particularly, ChatGPT can resolve more than 60% code style and maintainability issues in Python code with feedback from static analysis and runtime errors while more than 60% performance and efficiency issues in Java code can be addressed with a simple feedback.

> ***Finding 8:*** *ChatGPT shows great promise in self-mitigating code quality issues, achieving a fixed rate of 20% to 60%.*

In our comparison of two prompt designs, we observed that feedback with static analysis and runtime errors is more effective in fixing code style and maintainability while simple feedback performs better in the remaining quality issues in both Java and Python. This is because feedback from static analysis tools provides detailed information about code quality issues, guiding ChatGPT to self-mitigate these problems. For example, static analysis tools raise a warning that

```
1    Solution.java:12: ForLoopCanBeForeach: This for loop can be replaced by a
     foreach loop
```

for the initial solution in lines 1 in Code 5. The warning provides detailed information about the code style and maintainability issue in line 12 including location and even solution. Therefore, ChatGPT can easily mitigate the issue. Meanwhile, feedback with runtime errors for remaining issues, such as execution errors or performance and efficiency, tends to be less specific and more ambiguous. For example, in most of the performance and efficiency, we only obtain a “TIMEOUT" message, which does not reveal any details or root cause of a given issue. Similarly, for solution inaccuracies, the runtime errors also usually only contain an `AssertionError`. For example, in Code 4, ChatGPT has only received the following information from runtime errors:

```
1    AssertionError : Input : cost = [10 , _15_ ,20] Expected output : 15
```

Although the `AssertionError` points out the incorrect input-output examples, it remains abstract and does not provide precise guidance. As a result of such limited feedback, it is not surprising that ChatGPT shows lower performance in self-debugging issues. Interestingly, we found that simple feedback is more effective than static analysis feedback or runtime errors in resolving these issues. This is possibly due to the introduction of noise by static analysis and runtime error feedback, which can confuse ChatGPT and lead to incorrect patches.

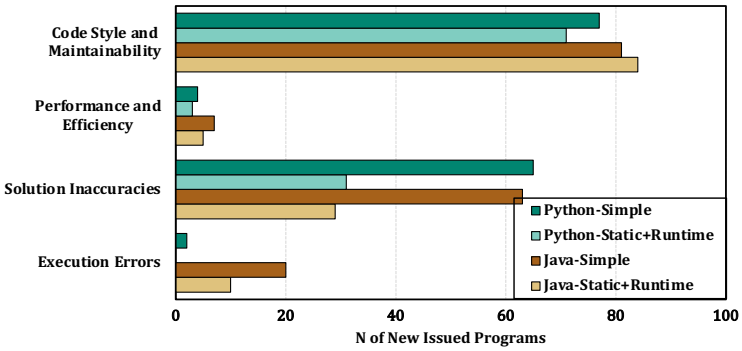

Fig. 5. Number of Newly Introduced Quality Issues after Providing Feedback

> ***Finding 9:*** *Prompts with detailed feedback can effectively assist ChatGPT in self-mitigating code quality issues, whereas ambiguous feedback may have a negative impact on ChatGPT's performance.*

While the aforementioned feedback are quite effective in guiding ChatGPT to self-mitigate its code quality issues, we still observe that new code quality issues are introduced in the fixed version. Figure 5 presents the number of newly introduced quality issues after providing feedback for both Java and Python. The results demonstrate that, in some cases, addressing certain code quality issues may inadvertently introduce new issues in the generated code. It can be seen that providing simple feedback results in a higher number of new issues compared to static feedback and runtime errors. This is because simple feedback is quite general so ChatGPT is sometimes misguided, resulting in introducing new problems though plausibly solving the original issues. We observed that ChatGPT sometimes even generates a new code, which is totally different from the original version.

Code 6. A example of newly-introduced issues in ChatGPT-generated fix using simple feedback for LeetCode Problem 1323-maximum-69-number

```
class Solution {
    //Initial solution with issues
    public int maximum69Number (int num) {
        int[] arr = String.valueOf(num).chars().map(c -> c-'0').toArray();
        for(int i=0; i<arr.length; i++){
            if(arr[i] == 6){
                arr[i] = 9;
                break;
            }
        }
        int res = 0;
        for(int i=0; i<arr.length; i++){
            res = res*10 + arr[i];
        }
        return res;
    }
    //Fixed by simple feedback
    public int maximum69Number (int num) {
        int digits = (int) Math.log10(num) + 1;
        int firstSixDigit = digits - 1;
        int temp = num;
```

```
22          while (temp > 0) {
23              if (temp % 10 == 6) {
24                  break;
25              }
26              temp = temp / 10;
27              firstSixDigit--;
28          }
29          return num + (int) Math.pow(10, firstSixDigit) * 3;
30      }
31      //Fixed by static analysis feedback
32      public int maximum69Number (int num) {
33          int[] arr = Integer.toString(num).chars().map(c -> c - '0').toArray();
34          for (int i = 0; i < arr.length; i++) {
35              if (arr[i] == 6) {
36                  arr[i] = 9;
37                  break;
38              }
39          }
40          int res = 0;
41          for (int digit : arr) {
42              res = res * 10 + digit;
43          }
44          return res;
45      }
46  }
```

For example, lines 18-30 in Code 6 show a fix generated by ChatGPT for the initial solution in lines 3-16. Unfortunately, instead of fixing the issue, ChatGPT generated a new solution (lines 18-30), which implement an incorrect solution, resulting in failing test cases. Static feedback and runtime errors, on the other hand, provide detailed information, i.e., leading to a correct fix in line 41 which change the for-loop to foreach-loop. The results show that providing more accurate feedback about code quality issues could lead to improvement in the quality of fixed programs by ChatGPT. These findings emphasize the importance of advanced feedback mechanisms and strategies that improve ChatGPT's self-debugging capabilities by reducing the introduction of new issues while resolving existing code quality problems.

> ***Finding 10:*** *Despite being effective in self-mitigating code quality issues, ChatGPT still introduces new code quality issues in the generated fixes. More precise feedback could help mitigate the issues.*

7 DISCUSSION

7.1 Lessons Learned

In this section, we highlight key lessons learned through our experiments and analysis that can drive future research in the field.

Code quality issues are prevalent in AI-generated code: Our study revealed that ChatGPT-generated code is prone to various code quality issues, including compilation and runtime errors, wrong outputs, and maintainability problems. This finding emphasizes the importance of addressing these issues to ensure the long-term success of AI-driven code generation and to maintain high-quality software systems.

Task difficulty, time that tasks are introduced, and program size impact automated code generation performance: We found that the performance of ChatGPT on code generation tasks

is significantly influenced by factors such as task difficulty, task-established time, and program size. This suggests that improvements in AI models should consider these factors to better adapt to different types of code generation tasks.

Tailored feedback and prompt engineering are crucial for effective self-debugging and code generation quality: Our results suggest that the effectiveness of ChatGPT's self-mitigating capabilities depends on the type of feedback provided, the programming language, and the specific code quality issue. For instance, static analysis feedback works better for code style and maintainability issues, while simple feedback is more effective for addressing execution errors and wrong outputs. This finding highlights the importance of providing tailored feedback to maximize the efficacy of ChatGPT's self-mitigating capabilities. Moreover, the quality of ChatGPT-generated code can be heavily affected by the choice of prompts. Future work could explore optimizing prompts to improve the accuracy and reliability of ChatGPT-generated code, further enhancing the overall effectiveness of AI-driven code generation models.

7.2 Threats to Validity

7.2.1 External validity. Threats to external validity concern the generalizability of our findings. Our study is based on a dataset of 2,033 programming tasks from LeetCode, which may not represent all possible code generation tasks encountered in real-world software development. Additionally, we focus on Java and Python, two popular programming languages; however, our findings may not be directly applicable to other programming languages. To mitigate these threats, future work could expand the dataset by incorporating tasks from various sources and diverse programming languages, and by considering different types of software projects, such as web applications, mobile apps, and embedded systems.

7.2.2 Internal validity. Threats to internal validity refer to possible errors in our experiments. One such threat relates to bugs happening in our code. To mitigate this risk, we have carefully checked our code and made our code publicly available in figshare. [2] Another possible threat may be introduced from our manual analysis and categorization. To eliminate the potential bias, we conducted a sorting discussion among three annotators. We also release our analysis and categorization results for public verification.

7.2.3 Construct validity. Threats to construct validity relate to the suitability of our evaluation. In our study, we use the pass@1 metric, in which a program is considered as functionally correct if it passes all the test cases. A possible threat arises from the incompleteness of the test suite, which could potentially result in missed program bugs. In our experiments, we use the original test suite from LeetCode, which is carefully designed and widely recognized. Thus, we believe this risk is minimal. Another potential threat to construct validity comes from the variability in ChatGPT-generated code due to different prompts. To address this concern, we followed the similar methodology used by Fan *et al.* [11] and Tian *et al.* [43], which ensures that our results are reliable by using a consistent set of prompts across different tasks. However, it is important to note that prompt engineering can significantly influence the quality of the generated code. Future work could focus on optimizing the prompts to improve the accuracy and reliability of ChatGPT-generated code, thus enhancing the overall effectiveness of AI-driven code generation models.

8 RELATED WORK

In this section, we present related work and discuss the novelty of our work with respect to large language models for code generation and code quality issues.

8.1 Large Language Model for Code Generation

Large Language Models have been emerging as the state-of-the-art in code-related tasks, advancing the progress on program understanding [12, 46], analysis [19, 22, 40], and generation [6, 18, 23]. Among these tasks, LLM-based code generation approaches are closely related to our study. The era of LLMs in code generation began with the introduction of CodeX [6], which serves as the backend for a well-known commercial tool, i.e., GitHub Copilot [14]. After the success of CodeX, various models such as InCoder [13], Google Alphacode [23], and Amazon CodeWhisperer [1] have been emerging, resulting in remarkable improvements in the effectiveness of code generation. These advancements provide a new way of tackling the code generation problem. Recently, OpenAI introduced ChatGPT [29] a general AI-powered chatbot with remarkable capabilities in language understanding and human-like answering. ChatGPT has shown remarkable accuracy in generating code and solving programming problems while receiving positive feedback from users and gaining popularity. However, the quality of the ChatGPT-generated code is the critical concern for top software companies when deciding whether to adopt it in practice or not [7, 16, 33].

To the best of our knowledge, this paper is the first to systematically analyze and characterize the code quality issues in ChatGPT-generated code. By doing so, we hope to increase awareness about the quality issues in code generated by ChatGPT, and provide suggestions for mitigating the issues.

8.2 Code Quality Issues

Code quality issues are the most important concern, as one quality issue (aka. software defect) could lead to monetary and reputation costs. There is a large number of studies investigating code quality issues from human-written code. For example, Kochhar *et al.* [21] conducted a large-scale empirical investigation on the code quality of open-source projects implemented in 17 programming languages. Saboury *et al.* [35] empirically investigate code smells in 537 releases of five popular Javascript applications. Keuning *et al.* [20] investigate code quality issues in student programs.

However, none of these studies focuses on the code quality issues for the AI-generated code, highlighting the difference between our paper and the literature on code quality issues.

8.3 Code Quality Issues of AI-generated Code

As AI-generated code becomes more prominent, researchers investigate the quality of code generated by CodeX and GitHub Copilot. For example, Nguyen *et al.* [27] evaluated the performance of Copilot on 33 programming tasks. Fan *et al.* [11] further analyze common bugs in code generated by Codex, the backend of Copilot, on 113 programming tasks and benchmark automated program repair tools on fixing the mistakes. However, these studies focus on CodeX and GitHub Copilot on a few programming tasks, leading to a lack of diversity. Different from these studies, we conduct a large-scale analysis on 2033 programming tasks, which enables us not only to comprehensively evaluate the effectiveness of AI-generated code but also to identify factors affecting their performance. Moreover, Nguyen *et al.* [27] focus on analyzing the correctness of AI-generated code while Fan *et al.* [11] target to benchmark automated program repair tools on fixing mistakes. Our work, on the other hand, delves deeper into analyzing code quality issues, including code style & maintainability issues, and highlights common patterns across different types of code quality problems. Besides that, our study focuses on ChatGPT, a recently-introduced AI chatbot developed by OpenAI. Unlike Codex and GitHub Copilot, which target experienced professional developers and programmers, ChatGPT caters to a much wider audience, with approximately 1.6 billion visits in April 2023 [37], including novice programmers and non-coders. Therefore, a comprehensive study on the reliability of source code generated by ChatGPT would not only provide valuable

insights into the model but also raise awareness about the responsible usage of ChatGPT in code generation.

9 CONCLUSION

In this study, we conducted a systematic analysis of ChatGPT-generated code to assess its reliability and identify potential code quality issues. Our findings demonstrate that while ChatGPT can generate functional code for various programming tasks, the generated code often suffers from quality issues, such as compilation and errors, wrong outputs, maintainability problems, and performance inefficiencies. We also explored ChatGPT's self-debugging capabilities and investigated the impact of different feedback strategies in addressing these code quality issues.

Our research provides valuable insights into the current limitations of ChatGPT and highlights the importance of considering context-aware feedback and code quality issues when utilizing AI-driven code generation tools. Moreover, our work offers insights for future research and development efforts aimed at enhancing the code generation capabilities of AI models like ChatGPT. We believe that by addressing these challenges, we can pave the way for more reliable, efficient, and maintainable AI-generated code, ultimately benefiting both experienced developers and novice programmers alike. In the future, we plan to develop more advanced prompts in both code generation and fixing to further improve the reliability of ChatGPT-generated code.

REFERENCES

[1] Amazon. 2023. *Amazon CodeWhisperer.* https://aws.amazon.com/codewhisperer/
[2] Anonymous Author. 2023. *Replication Package for Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues.* https://doi.org/10.6084/m9.figshare.22773665
[3] Lingfeng Bao, David Lo, Xin Xia, Xinyu Wang, and Cong Tian. 2016. How Android app developers manage power consumption? An empirical study by mining power management commits. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 37–48.
[4] Oliver Burn. 2003. Checkstyle. *http://checkstyle. sourceforge. net/* (2003).
[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).
[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[7] Jim Chilton. 2023. *The New Risks ChatGPT Poses to Cybersecurity.* https://hbr.org/2023/04/the-new-risks-chatgpt-poses-to-cybersecurity
[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
[9] Tom Copeland. 2005. *PMD applied*. Vol. 10. Centennial Books San Francisco.
[10] Ian Cordasco and Tarek Ziade. 2010. Flake8: Your tool for style guide enforcement. *Programa de computador* (2010).
[11] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *the 45th International Conference on Software Repositories (ICSE)*.
[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.
[13] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
[14] GitHub. 2023. *GitHub Copilot.* https://github.com/features/copilot
[15] Tanya Goyal, Junyi Jessy Li, and Greg Durrett. 2022. News summarization and evaluation in the era of gpt-3. *arXiv preprint arXiv:2209.12356* (2022).
[16] Morey Haber. 2023. *Two Cybersecurity Concerns When Using ChatGPT For Software Development.* https://www.forbes.com/sites/forbestechcouncil/2023/03/29/two-cybersecurity-concerns-when-using-chatgpt-for-software-development

[17] Amr Hendy, Mohamed Abdelrehim, Amr Sharaf, Vikas Raunak, Mohamed Gabr, Hitokazu Matsushita, Young Jin Kim, Mohamed Afify, and Hany Hassan Awadalla. 2023. How good are gpt models at machine translation? a comprehensive evaluation. *arXiv preprint arXiv:2302.09210* (2023).

[18] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[19] Milod Kazerounian, Jeffrey S Foster, and Bonan Min. 2021. SimTyper: sound type inference for Ruby using type equality prediction. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[20] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 110–115.

[21] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 563–573.

[22] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. 2022. AutoPruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 520–532.

[23] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[24] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 415–425.

[25] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[26] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[27] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*.

[28] Carlini Nicholas, Ippolito Daphne, Jagielski Matthew, Lee Katherine, Tramer Florian, and Zhang Chiyuan. 2023. Quantifying Memorization Across Neural Language Models. In *The Eleventh International Conference on Learning Representations*. 70–80.

[29] OpenAI. 2022. *Introducing ChatGPT*. https://openai.com/blog/chatgpt

[30] OpenAI. 2023. *ChatGPT Release Notes*. https://help.openai.com/en/articles/6825453-chatgpt-release-notes

[31] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[32] OpenAI. 2023. *Model Index For Researchers*. https://platform.openai.com/docs/model-index-for-researchers

[33] Carly Page. 2023. *Is ChatGPT a cybersecurity threat?* https://techcrunch.com/2023/01/11/chatgpt-cybersecurity-threat/

[34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[35] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in javascript projects. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 294–305.

[36] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[37] SimilarWeb. 2023. *ChatGPT's traffic overview*. https://www.similarweb.com/website/chat.openai.com

[38] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[39] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* 33 (2020), 3008–3021.

[40] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 481–496.

[41] TheGuardian. 2022. *ChatGPT reaches 100 million users two months after launch*. https://www.theguardian.com/technology/2023/feb/02/chatgpt-100-million-users-open-ai-fastest-growing-app

[42] Sylvain Thénault et al. 2001. Pylint. *Code analysis for Python* (2001).

[43] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[44] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. 2020. Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 327–337.

[45] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 413–424.

[46] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[47] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1480–1496.

[48] Hu Xu, Bing Liu, Lei Shu, and Philip Yu. 2020. DomBERT: Domain-oriented Language Model for Aspect-based Sentiment Analysis. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1725–1731. https://doi.org/10.18653/v1/2020.findings-emnlp.156

[49] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. 2020. Sentiment analysis for software engineering: How far can pre-trained transformer models go?. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 70–80.