2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-5388-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSRE62328.2024.00030

Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot

Shuang Li
School of Computer Science
Wuhan University
Wuhan, China
shuangli.cs@whu.edu.cn

Yuntao Cheng
School of Computer Science
Wuhan University
Wuhan, China
cytzzz@whu.edu.cn

Jinfu Chen*
School of Computer Science
Wuhan University
Wuhan, China
jinfuchen@whu.edu.cn

Jifeng Xuan*
School of Computer Science
Wuhan University
Wuhan, China
jxuan@whu.edu.cn

Sen He
Department of Systems and Industrial Engineering
University of Arizona
Tucson, United States
senhe@arizona.edu

Weiyi Shang
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
wshang@uwaterloo.ca

***Abstract*—The integration of Large Language Models (LLMs) into software development tools like GitHub Copilot holds the promise of transforming code generation processes. While AI-driven code generation presents numerous advantages for software development, code generated by large language models may introduce challenges related to security, privacy, and copyright issues. However, the performance implications of AI-generated code remain insufficiently explored. This study conducts an empirical analysis focusing on the performance regressions of code generated by GitHub Copilot across three distinct datasets: HumanEval, AixBench, and MBPP. We adopt a comprehensive methodology encompassing static and dynamic performance analyses to assess the effectiveness of the generated code. Our findings reveal that although the generated code is functionally correct, it frequently exhibits performance regressions compared to code solutions crafted by humans. We further investigate the code-level root causes responsible for these performance regressions. We identify four major root causes, i.e., inefficient function calls, inefficient looping, inefficient algorithm, and inefficient use of language features. We further identify a total of ten sub-categories of root causes attributed to the performance regressions of generated code. Additionally, we explore prompt engineering as a potential strategy for optimizing performance. The outcomes suggest that meticulous prompt designs can enhance the performance of AI-generated code. This research offers valuable insights contributing to a more comprehensive understanding of AI-assisted code generation.**

***Index Terms*—Code generation, Software performance, Github Copilot, Program analysis**

I. Introduction

The integration of large language models (LLMs) into software development tools has introduced a new era of AI-powered coding assistants. These LLM-based tools, such as GitHub Copilot, ChatGPT, and CodeWhisperer, are redefining how developers write code. One typical example is GitHub Copilot, a tool that leverages LLMs to aid programmers by suggesting code completions and functionalities. The LLM-based code generation tools offer the potential to enhance developer productivity and streamline development processes.

While Copilot offers the potential to enhance developer productivity, ensuring the quality of the generated code remains a crucial area of investigation. Prior research has extensively studied and reported challenges related to correctness [35, 56, 57], security [40, 16, 28], and privacy [37, 25, 55] associated with code generated by LLMs. These studies highlight the need for continuous improvement in the overall quality of LLM-generated code. However, the performance implications of AI-generated code are a critical yet unexplored area.

Performance is a critical aspect of software quality. Software performance regressions may affect application responsiveness, resource consumption, and overall user experience. The efficiency of code can be particularly crucial in performance-sensitive domains such as high-frequency trading, real-time systems, and large-scale data processing. Given the potential for AI-generated code to either enhance or degrade performance, it is imperative to evaluate its performance characteristics, including the risk of performance regressions. There is a gap in understanding whether these AI assistants can facilitate the generation of high-performing code.

To fill this gap, we design an experimental setup that involves generating code using GitHub Copilot and evaluating its performance regressions using both static analysis tools and dynamic profiling. To ensure the broad applicability of our findings, we select three diverse and representative datasets, i.e., HumanEval, AixBench, and MBPP. The static analysis is supported by tools such as Qodana, Spotbugs, and PMD, which are adept at identifying a variety of performance regression code issues. For dynamic analysis, we choose cProfile, Memory-profiler, and Psutil to measure critical performance metrics such as runtime, memory usage, and CPU utilization.

Our study finds that AI-generated code, although functionally correct, often exhibits performance regressions when compared to canonical solutions. We identify several root

*Corresponding authors.

causes that contribute to these regressions, including inefficient function calls, suboptimal looping constructs, and inefficient use of language features. Furthermore, we demonstrate that prompt engineering can be an effective technique to reduce the performance regressions of AI-generated code.

To facilitate research reproducibility, we make the original datasets and scripts available in our replication package [3]. Our contributions are summarized below:

- **Performance assessment of Copilot-generated code:** We assess the performance regressions of code generated by GitHub Copilot compared to human-written solutions.
- **Performance regression root causes of Copilot-generated code:** We qualitatively analyze the root causes of performance regression in the Copilot-generated code, providing valuable insights into potential shortcomings of current AI-powered coding assistants.
- **Prompt engineering for performance in Copilot-generated code:** We explore the usage of prompt engineering, a technique where developers tailor instructions provided to Copilot, to optimize the performance of generated code.

II. Background and Motivating Example

A. *GitHub Copilot*

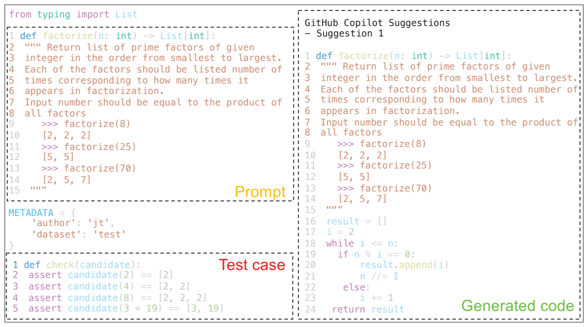
GitHub Copilot is an AI-assisted programming tool that enhances developer productivity by providing code generation services. GitHub Copilot empowers programmers by offering various forms of code completion. This functionality can be particularly beneficial in scenarios where developers have a clear understanding of the desired outcome but require assistance in translating that concept into functional code. Copilot offers two primary methods for code completion:

- Developers can select a specific section of code and request Copilot to automatically complete it. This functionality leverages the surrounding code context to generate relevant suggestions.
- Developers can use natural language comments to describe their desired functionality. Copilot then analyzes these comments and suggests code that aligns with the described requirements.

Copilot supports a variety of popular programming languages, including Python, JavaScript, TypeScript, Ruby, Go, and Java [18]. In this study, we leverage Copilot's capabilities to generate code for datasets encompassing two specific languages: Python and Java. This focus allows us to conduct a focused analysis of the performance implications of AI-generated code within these widely used languages.

B. *A motivating example of using Copilot to generate code*

While GitHub Copilot offers significant potential for developers, a critical aspect to consider is the performance of the generated code. Here, we present a motivating example highlighting this challenge. One wants to develop a function to determine if a given integer is a prime number. To achieve this, one uses GitHub Copilot within the Visual Studio Code [19]



```
from typing import List
def factorize(n: int) -> List[int]:
 """ Return list of prime factors of given
 integer in the order from smallest to largest.
 Each of the factors should be listed number of
 times corresponding to how many times it
 appears in factorization.
 Input number should be equal to the product of
 all factors
    >>> factorize(8)
    [2, 2, 2]
    >>> factorize(25)
    [5, 5]
    >>> factorize(70)
    [2, 5, 7]
  """

METADATA = {
    'author': 'jt',
    'dataset': 'test'
}

def check(candidate):
 assert candidate(2) == [2]
 assert candidate(4) == [2, 2]
 assert candidate(8) == [2, 2, 2]
 assert candidate(3 * 19) == [3, 19]
```

```
GitHub Copilot Suggestions
- Suggestion 1

def factorize(n: int) -> List[int]:
 """ Return list of prime factors of given
 integer in the order from smallest to largest.
 Each of the factors should be listed number of
 times corresponding to how many times it
 appears in factorization.
 Input number should be equal to the product of
 all factors
    >>> factorize(8)
    [2, 2, 2]
    >>> factorize(25)
    [5, 5]
    >>> factorize(70)
    [2, 5, 7]
  """
  result = []
  i = 2
  while i <= n:
    if n % i == 0:
        result.append(i)
        n //= I
     else:
        i += 1
   return result
```

Fig. 1: An example of generating the function to determine all prime factors of a given number from HumanEval-25.py.

environment to generate the function. The generated code, while functionally correct, exhibits performance regressions. As illustrated in Figure 1, the generated code employs a naive approach that iterates from 2 to the original number, leading to inefficiency. This example emphasizes the potential for AI-generated code to introduce performance regressions. While Copilot can produce functional code, it may not always prioritize optimal performance. Our study aims to bridge this gap by analyzing the performance characteristics of code generated by GitHub Copilot. By understanding these characteristics, we can develop best practices and techniques to optimize performance and unlock the full potential of AI-powered coding assistants.

III. Case Study Setup

A. *Dataset*

Evaluating the quality and effectiveness of large language models (LLMs) in code generation requires specialized datasets designed to assess correctness and executability. These datasets typically include unique identifiers (IDs), natural language descriptions, function names (or specifications), and corresponding test cases. Our study explores three widely used datasets for code generation evaluation, focusing on both Python and Java languages:

- **HumanEval:** This handcrafted Python dataset by Chen et al. consists of 164 problems [7]. Each problem provides a function name, function body, associated test cases, and canonical solution. These problems focus on core programming skills like semantic understanding, algorithm design, and basic math.
- **MBPP:** MBPP is a Python dataset containing 974 problems [4]. Each problem is presented with a brief description and corresponding test cases.
- **Aixbench:** Designed for Java code generation, Aixbench [22] offers 187 problems, along with function signatures and test cases.

The overview of datasets is shown in Table I. HumanEval and MBPP datasets are chosen for their broad acceptance within the research community, as highlighted by Zheng et al. [62] and Zan et al. [60]. These datasets are widely recognized for their robustness and relevance in code generation and performance evaluation. The canonical solutions are

TABLE I: Overview of datasets used in our study

Dataset	Language	#Instances	Year	Reference
HumanEval	Python	164	2021	[7]
AixBench	Java	187	2022	[22]
MBPP	Python	974	2021	[4]

authored by senior developers, representing high-quality and efficient solutions for specific problems, as documented by prior studies [7, 4]. This selection provides a solid foundation for benchmarking AI-generated code against well-established standards and effectively assessing performance regression.

B. Experimental setup

In this subsection, we describe the process of collecting the generated code and performance data for each question in our study datasets. Figure 2 illustrates the overall process of our approach. We follow four steps to collect the needed data. In the first step, we prepare prompts by parsing the three datasets, i.e., HumanEval, AixBench, and MBPP. In the second step, for each prepared prompt, we feed the prepared prompt to GitHub Copilot to generate code. In the third step, we filter the generated code using test cases. Finally, we analyze the performance regressions of the generated code.

Step 1: Preparing prompt. The data from HumanEval, MBPP, and Aixbench datasets are stored in JSON files. We first parse these files to extract relevant code information. New files are created in either *.py* or *.java* format for each code snippet requiring completion. As an example in Figure 3, the *4.py* file is extracted from the corresponding HumanEval JSON file. The extracted function name, parameters, and comments are used as the prompt for Copilot.

Step 2: Generating code. In this step, we use Copilot to generate code for each prompt of each question. In detail, we leverage GitHub Copilot within the VSCode [50] environment to generate code for each prompt. We open each newly created file, activating the command panel with *Ctrl+Enter*, and Copilot provides 1-10 code completion suggestions. We consistently accept the first suggestion (see Figure 1) for consistency. This process is repeated for all files requiring completion across all datasets. Generated code files are named sequentially, i.e., HumanEval-0 to -163, MBPP-1 to -974, and AixBench-0 to -186.

Step 3. Filtering generated code. In this step, we filter the generated code from the last step by executing the corresponding test case. We execute the corresponding test cases on each generated code to evaluate correctness. Code that compiles successfully and passes the tests is retained. This filtering process resulted in correctness rates of 81.7% (HumanEval), 87.9% (MBPP), and 50.3% (Aixbench).

Step 4. Analyzing performance regressions. To comprehensively evaluate the performance regressions of generated code, we employ both static and dynamic analyses to examine the aforementioned filtered generated code. Static analysis is used to identify factors that may lead to performance regressions, while dynamic analysis observes differences in runtime, CPU usage, and memory consumption.

1) Static performance regression analysis: We use three static analysis tools: Qodana [45], Spotbugs [48], and PMD [41] to investigate potential performance regression issues in the generated code. Qodana is designed for Python programs. It integrates with CI processes and provides in-depth inspections across multiple languages, identifying errors, code smells, and standard violations. Spotbugs is an open-source tool for Java, which focuses on detecting bugs and vulnerabilities, with an emphasis on runtime errors and non-standard practices. PMD can identify potential flaws and complexity issues, promote code style consistency, and focus on optimization opportunities.

Since built-in rules in the three tools might not comprehensively cover performance regression, we develop custom rules based on extensive literature research and industry documentation. We use keywords such as “performance degradation”, “performance regression”, “anti-pattern”, and “code smell” to search in ACM Digital Library, IEEE Xplore Digital Library, Springer Link Digital Library, and Google Scholar. Based on the relevant literature retrieved from the above paper databases, we manually filter and obtain the code performance regression rules. In addition, we find some code performance regression rules in industrial documentation such as SonarQube [47]. This search result covers anti-patterns, code smells, and predefined PMD rules related to performance. The customized rules are categorized into six aspects: Performance Regression, Bad Practice, Dodgy Code, Error Prone, Bad Design, and Multithreading, to provide a structured approach to identifying performance regression. Finally, we identify a total of 159 rules related to performance regressions, i.e., “Manually copying data between two arrays is inefficient. Please use a more efficient native array assignment method instead” [41].

For Spotbugs, we use its Idea-based plugin. Initially, we open a given project in Idea for inspection. Subsequently, we scan the project using the Spotbugs plugin. We can then obtain the performance regression results of Spotbugs detection. For PMD, we first create a new XML file and include all custom rules within the `<ruleset>` element. We then define a `<rule>` element for each rule and set various attributes for the rule within this element. Next, we write XPath expressions or Java classes to implement the matching logic for corresponding rules. For Qodana, we first create new projects on Qodana Cloud. We then upload the HumanEval and MBPP datasets to projects. In this way, Qodana can automatically identify and highlight potential performance regression issues.

2) Dynamic performance regression analysis: Dynamic analysis is conducted on Python datasets from HumanEval and MBPP, focusing on runtime, memory usage, and CPU utilization. We conduct this analysis using three profiling tools:

cProfile [1]: A Python library for performance analysis, providing metrics like the number of function calls and time spent in functions.

Memory-profiler [14]: A module for tracking memory consumption, allowing for the decoration of functions to monitor memory usage.

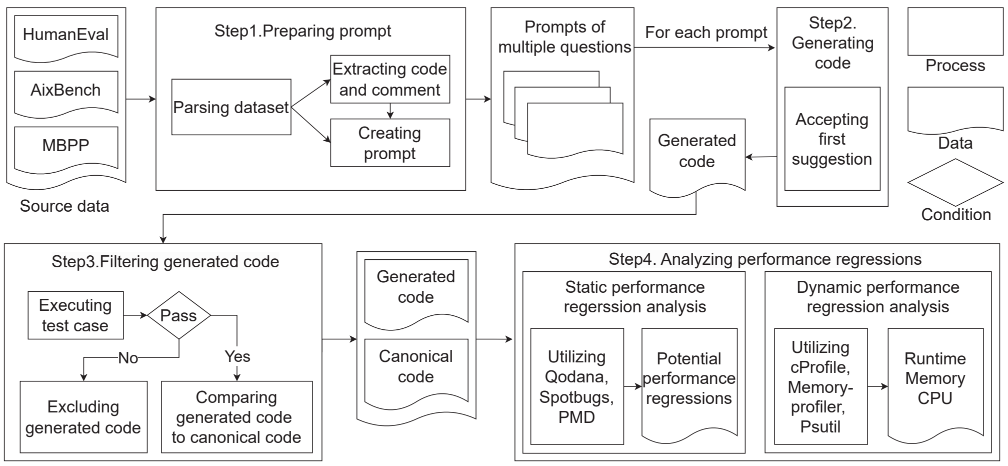
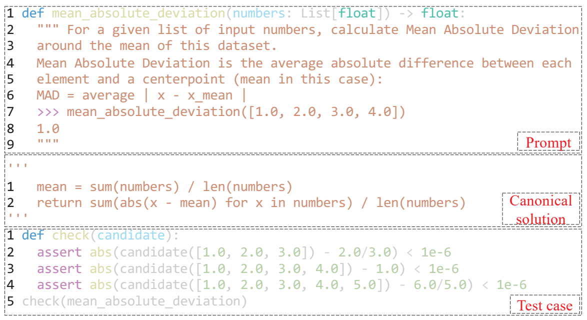

HumanEval
AixBench
MBPP
Source data
Step1.Preparing prompt
Parsing dataset
Extracting code and comment
Creating prompt
Prompts of multiple questions
For each prompt
Step2. Generating code
Accepting first suggestion
Generated code
Process
Data
Condition
Step3.Filtering generated code
Executing test case
Pass
No
Yes
Excluding generated code
Comparing generated code to canonical code
Generated code
Canonical code
Step4. Analyzing performance regressions
Static performance regerssion analysis
Utilizing Qodana, Spotbugs, PMD
Potential performance regressions
Dynamic performance regression analysis
Utilizing cProfile, Memory-profiler, Psutil
Runtime Memory CPU

Fig. 2: An overview of our approach to collecting data.


```
def mean_absolute_deviation(numbers: List[float]) -> float:
  """ For a given list of input numbers, calculate Mean Absolute Deviation
  around the mean of this dataset.
  Mean Absolute Deviation is the average absolute difference between each
  element and a centerpoint (mean in this case):
  MAD = average | x - x_mean |
  >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
  1.0
  """
```

```
'''
  mean = sum(numbers) / len(numbers)
  return sum(abs(x - mean) for x in numbers) / len(numbers)
'''
```

```
def check(candidate):
  assert abs(candidate([1.0, 2.0, 3.0]) - 2.0/3.0) < 1e-6
  assert abs(candidate([1.0, 2.0, 3.0, 4.0]) - 1.0) < 1e-6
  assert abs(candidate([1.0, 2.0, 3.0, 4.0, 5.0]) - 6.0/5.0) < 1e-6
check(mean_absolute_deviation)
```

Fig. 3: Examples of prompt preparation, corresponding canonical solution, and the test case from the HumanEval dataset.

Psutil [15]: A library for process and system utilization information, particularly useful for CPU monitoring.

The dynamic analysis involves running the generated code and collecting data on its performance metrics. This data will be used to evaluate the performance characteristics of the code. Each tool will target a specific aspect of performance, with cProfile for runtime, memory-profiler for memory, and Psutil for CPU usage. The hardware configuration for the experiment includes an Intel Core i9-13900K processor, 128 GB of RAM, 1 TB SSD for primary storage, 8 TB HDD for secondary storage, and the system operates on Ubuntu 22.04.4 LTS. By following the experimental setup, we aim to achieve a thorough and systematic evaluation of the performance regressions of AI-generated code, considering both static and dynamic aspects, and providing actionable insights for code optimization and tool improvement.

IV. Case Study Results

We present our case study results by addressing three research questions, covering motivation, approach, and findings.

A. RQ1: How prevalent are performance regressions in generated code by Copilot?

Motivation: While prior research has focused on evaluating the correctness and security of AI-generated code, performance regression has received less attention. However, intuitively, AI-generated code models may not fully grasp the developer's performance goals, potentially leading to code that prioritizes functionality over efficiency. On the other hand, training data for code generation models might not explicitly emphasize performance considerations, impacting the models' ability to generate efficient code. Given these potential shortcomings, it's crucial to investigate the prevalence of performance regressions in AI-generated code. Understanding the scope of this issue will inform future research directions and development efforts for AI-assisted coding tools.

Approach: Our approach involves two strategies to evaluate the performance of code generated by GitHub Copilot. In particular, for static performance regression analysis, we employ industry-standard tools, i.e., Spotbugs and PMD, to scan the generated Java code in the AixBench dataset. These tools are equipped with pre-defined rules that can detect performance regressions within the code. The detailed configuration of these rules is available in the replication package we provided. For Python code in the HumanEval and MBPP datasets, we use Qodana, a cloud-based static analysis platform to identify potential performance regressions specific to Python code. To facilitate efficient analysis, we create new projects and establish a dedicated scan workflow within Qodana Cloud. This workflow enables Qodana to automatically identify and highlight potential performance-related code issues within the generated Python code.

For dynamic performance regression analysis, we compare the generated code with canonical solutions from the HumanEval and MBPP datasets. Using the dynamic performance regression detection modules, we conduct dynamic performance regression analysis on these two datasets' generated and canonical code sets. The Python scripts generated for the HumanEval and MBPP datasets are typically short and have brief single-run execution times. To generate more robust performance data, we adopt a technique called repetitive iteration measurement [29, 10, 27]. This technique extends the runtime of the generated code by increasing the number of iterations within the test cases, allowing profiling tools to capture more comprehensive performance data. We achieve

```
def check(candidate):
    for item in range(30):
        assert abs(candidate([1.0, 2.0, 3.0]) - 2.0/3.0) < 1e-6
        assert abs(candidate([1.0, 2.0, 3.0, 4.0]) - 1.0) < 1e-6
        assert abs(candidate([1.0, 2.0, 3.0, 4.0, 5.0]) - 6.0/5.0) < 1e-6
```

Fig. 4: An example of increasing the number of iterations within the test case of Figure 3.

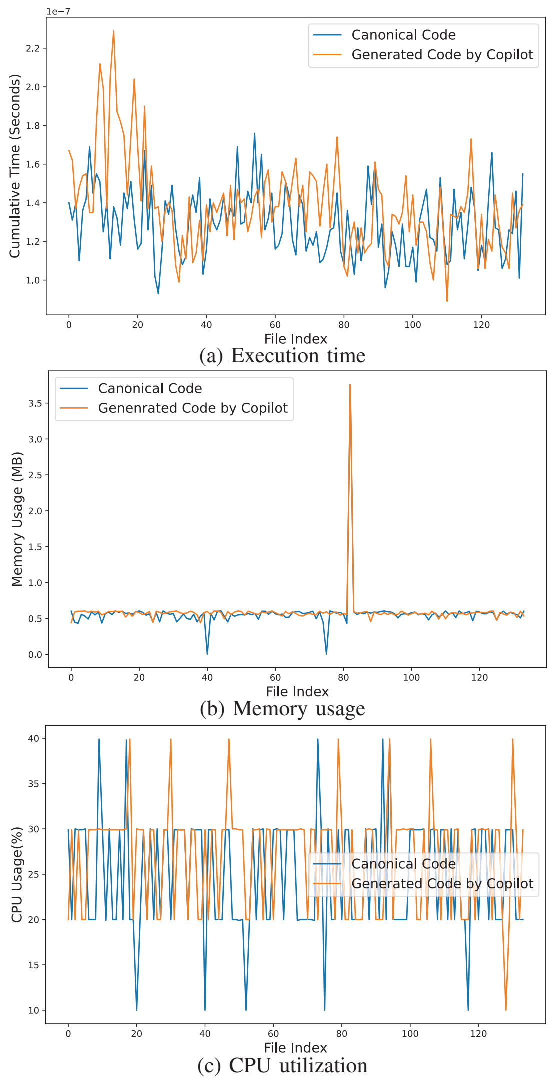
this extension by adding a for loop at the beginning of the test cases. This loop causes the existing test cases to be executed repeatedly. Figure 4 illustrates this modification for the script shown in Figure 3. Once the iterations has been increased, we encapsulate each script from the HumanEval and MBPP datasets within a function. We then use the cProfile module to profile these functions. Extracting the "cumtime" metric from the profiling results reveals the script's overall runtime.

We use both domain-level performance metrics, i.e., execution time, and physical-level performance metrics, i.e., CPU and memory usage, as measurements of performance regressions. To monitor memory usage during script execution, we add the "@profile" decorator at the beginning of each function. This decorator activates the "memory_usage" function, which collects the memory footprint of the function's execution. Similarly, we leverage the "cpu_percent" function from the psutil to capture CPU usage during script execution.

Results: Performance regressions are not rare instances in code generated by Copilot. We detect 8 and 38 suspicious low-performing code snippets in Spotbugs and PMD results, respectively. These findings suggest a high likelihood of performance regressions within the generated Java code. For the Python code in the HumanEval and MBPP datasets, Qodana identifies 14 and 274 instances of potential performance regressions, respectively.

In terms of dynamic performance regression analysis, we observe that while the code generated by Copilot occasionally outperforms the canonical code, it typically exhibits a substantial performance regression. We define a performance regression as significant if the generated code's performance is over 20% worse than the canonical code's performance [59]. In the HumanEval dataset, 31 out of 134 scripts show significant runtime discrepancies (see Figure 5). We observe 14 scripts with substantial memory usage disparities and 54 with notable CPU utilization gaps in the HumanEval dataset. This results in 79 scripts exhibiting significant regression in at least one performance metric when compared to the canonical code. The MBPP dataset presents a different set of disparities. 226, 14, and 265 scripts contain performance regression in runtime, memory usage, and CPU utilization, respectively. Table II presents the performance regressions observed in these datasets, showing the number of notable performance regressions among scripts that passed the test cases.

Physical performance metrics are important complementary indicators of performance regressions in generated code. We use the two physical performance metrics, i.e., CPU utilization and memory usage, to measure performance regression. Our findings indicate that when considering physical performance metrics, we are able to identify additional instances of performance regression that might have



(a) Execution time

(b) Memory usage

(c) CPU utilization

Fig. 5: Performance comparison between generated code and canonical code for each prompt in the HumanEval dataset.

been overlooked if we had relied solely on execution time. Specifically, within the HumanEval dataset, we find 39 and 11 code instances exhibiting performance regressions in terms of CPU utilization and memory usage even though their execution time seemed acceptable. These findings emphasize the importance of considering a multi-faceted approach to performance evaluation during code generation.

The performance regressions identified in code generated by Copilot can have broader implications for software systems. Our findings underscore the need for careful evaluation of AI-generated code, especially in performance-critical applications. The findings suggest the need for more frequent performance assurance activities (like performance testing) in practice.

TABLE II: Number of identified performance regression in code generated by Copilot

		Static performance regression analysis			Dynamic performance regression analysis		
Dataset	#Passed instances (percentage)	SpotBugs	PMD	Qodana	Execution time	Memory usage	CPU Utilization
HumanEval	134 (81.7%)	N/A	N/A	14	31	14	54
MBPP	856 (87.9%)	N/A	N/A	274	226	14	265
AixBench	88 (50.3%)	8	38	N/A	N/A	N/A	N/A

B. RQ2: What are the root causes in Copilot-generated code that lead to performance regression?

Motivation: In RQ1, we find that there are prevalent performance regressions in the code generated by Copilot. While identifying these regressions is crucial, a more profound understanding of the root causes is essential for mitigating their impact. By pinpointing the reasons and patterns that lead to performance regressions, we can provide valuable insights that can inform the development of Copilot itself, potentially guiding the AI model toward generating more performant code. We can also inform developers with guidance on how to recognize potential performance pitfalls during the code generation process with Copilot.

Approach: To investigate the underlying root causes responsible for performance regressions in Copilot-generated code, we employ a qualitative research approach known as open coding. This method involves manually examining code samples, allowing us to uncover root causes that contribute to performance regressions. We recognize that directly analyzing code root causes can introduce potential subjectivity and bias. To mitigate this concern, we follow a rigorous process inspired by prior research methodologies [61, 10]:

- **Dual coding.** Two authors independently analyze a consistent set of generated code instances with identified performance regressions of at least one performance metric, alongside their canonical code. We identify 24, 29, and 266 unique code instances with performance regression in HumanEval, AixBench, and MBPP, respectively. This step is critical for surfacing any inconsistencies in the interpretation of code root causes.
- **Disagreement resolution.** When discrepancies arise between the initial analyses, a third author facilitates a discussion. This collaborative review refines and aligns the identified root causes for performance regressions.
- **Iterative analysis.** The examination is repeated in an iterative process until no new root causes of code-related performance regressions are discovered, indicating that a comprehensive understanding of the prevalent root causes has been achieved.

To quantify the reliability of our dual coding analysis, we calculate the Cohen's Kappa statistic, which yielded a considerable agreement score of 0.88 [34].

Results: We identify four major root causes of performance regressions from the code generated by Copilot. Our in-depth analysis of code samples exhibiting performance regressions yields four key categories of root causes at the code level, along with ten sub-categories. These categories are detailed in Table III. Below, we discuss each root cause category with corresponding code examples for illustration.

```
for (int i = array.length - 1; i > 0; i--) {
    int rand = (int) (Math.random() * (i + 1));
    int temp = array[i];
    array[i] = array[rand];
    array[rand] = temp;
}
```

(a) Inefficient API Usage from AixBench-11

```
def is_simple_power(x, n):
    if x == 1:
        return True
    if x < n:
        return False
    if x % n != 0:
         return False
    return is_simple_power(x // n, n)
```

(b) Excessive Recursion from HumanEval-76

Fig. 6: Examples of inefficient function calls.

*1) **R1 Inefficient Function Calls**:* Performance regression is often attributable to suboptimal function call choices, including the use of inefficient APIs and excessive recursion leading to deep stack issues.

R1-1 Inefficient API Usage. Selecting the correct functions and APIs has a significant impact on performance. The generated code often opts for less efficient functions when more optimal methods are available to enhance efficiency. For example, in the code snippet shown in Figure 6 (a) from AixBench-11, the generated code uses the *Math.random()* API to generate random double numbers and then convert them to int type. This approach is less efficient in terms of performance compared to directly using the *random.nextInt()* API.

R1-2 Excessive Recursion. Recursive functions are used in the generated code. In code implementation, employing recursive functions can lead to excessive stack depth when handling large data ranges, thus impacting performance. For example, in the code shown in Figure 6 (b) from HumanEval-76, the function *is_simple_power(x, n)* is designed to check if the number x is the power of another number n. However, this recursive function may suffer from performance regression due to increased recursion depth when faced with very large values of x or when n is close to 1.

*2) **R2 Inefficient Looping**:* Performance regressions are frequently linked to inefficient looping. For instance, operations such as string concatenation within loops, multiple nested loops, and object creation inside loops can lead to significant performance declines. These root causes particularly affect the efficiency of code when handling large amounts of data.

R2-1 String Concatenation in Loops. Strings are immutable, and using the + operator to concatenate strings creates a new string object. Performing such operations frequently within a loop, especially when n is large, can lead to substantial memory allocation and release, thus potentially reducing the efficiency of code execution. An example of this

TABLE III: Root causes of performance regression from the Copilot-generated Code

Dataset	Inefficient Function Calls			Inefficient Looping				Inefficient Algorithm			Inefficient Use of Language Features			
	Inefficient API usage	Excessive Recursion	ALL	String Concatenation in Loops	Nested Looping	Object Creation in Loops	ALL	Missed Mathematical Optimizations	Suboptimal Conditional Logic	ALL	Underutilization of Language Features	Unused Variables	Inefficient Exception Handling	ALL
HumanEval	8	1	9	2	4	1	7	2	1	3	5	0	0	5
AixBench	9	1	10	1	4	1	6	1	1	2	3	1	7	11
MBPP	53	6	59	16	40	10	66	53	23	76	54	6	5	65

```python
def string_sequence(n: int) -> str:
    result = ’’
    for i in range(n + 1):
        result += str(i) + ‘ ’
    return result[:-1]
```
(a) String Concatenation in Loops from HumanEval-15

```python
def sort_matrix(M):
    n = len(M)
    for i in range(n):
        for j in range(n-i-1):
            if sum(M[j]) > sum(M[j+1]):
                M[j], M[j+1] = M[j+1], M[j]
     return M
```
(b) Nested Looping from MBPP-12

```python
def add_K_element(test_list, K):
    res = []
    for i in test_list:
        temp = []
        for j in i:
            temp.append(j+K)
    res.append(tuple(temp))
    return (res)
```
(c) Object Creation in Loops from MBPP-363

Fig. 7: Examples of inefficient looping.

```python
# Generated code            # Canonical solution
def ap_sum(a,n,d):          def ap_sum(a,n,d):
  total = 0                   total = (n * (2 * a + (n - 1) * d)) / 2
  for i in range(n):          return total
    total += a + i * d
  return total
```
(a) Missed Mathematical Optimizations from MBPP-335

```python
# Generated code                                        # Canonical solution
def greatest_common_divisor(a:int,b:int)->int:          while b:
  while a != b:                                           a, b = b, a % b
    if a > b:                                           return a
      a -= b
    else:
      b -= a
  return a
```
(b) Suboptimal Conditional Logic from HumanEval-13

Fig. 8: Examples of inefficient algorithm.

code is shown in Figure 7 (a) from HumanEval-15.

R2-2 Nested Looping. The generated code exhibits issues with nested loops. In the code snippet shown in Figure 7 (b) from MBPP-12, a double loop is used for comparing and swapping rows of a matrix. This nested looping structure, when handling larger matrices, leads to a time complexity of $O(n^2)$, resulting in significant performance regression.

R2-3 Object Creation in Loops. Creating objects repeatedly within loops can negatively impact performance due to extensive memory allocation and frequent garbage collection. In Figure 7 (c) from MBPP-363, the code example demonstrates a performance regression, where a new temp list is created within each iteration of a loop.

*3) **R3 Inefficient Algorithm**:* Our analysis finds that the generated code may employ algorithms that are less efficient than canonical solutions. In particular, we identify two subcategories of inefficient algorithms.

R3-1 Missed Mathematical Optimizations. The generated code may not always leverage mathematical optimizations to their full potential. In Figure 8 (a) from MBPP-335, the generated code iterates over the array through the loop and accumulates them one by one. However, from a mathematical perspective, the sum of an arithmetic sequence can be directly calculated using a formula, which is more efficient and avoids unnecessary loops.

R3-2 Suboptimal Conditional Logic. The use of complex or unnecessary conditional statements (if-else) can introduce performance regression. As shown in Figure 8 (b) from HumanEval-13, when calculating the greatest common divisor (GCD), the generated code implementation reduces the difference between the two numbers by repeatedly subtracting the smaller number from the larger one, increasing the runtime. This inefficient conditional logic results in unnecessary performance regression.

*4) **R4 Inefficient Use of Language Features**:* The generated code may exhibit shortcomings in performance in its utilization of the programming language's built-in features and functionalities.

R4-1 Underutilization of Language Features. Sometimes the generated code fails to effectively leverage the features of the programming language, as demonstrated in Figure 9 (a) from MBPP-688. When calculating the magnitude of a complex number, the generated code manually computes it, not fully utilizing Python's built-in capabilities for handling complex numbers. Using Python's *cmath* module or the built-in complex type and abs function could offer performance benefits, as built-in operations are typically closer to the hardware level and more optimized.

R4-2 Unused Variables. When generating code, Copilot sometimes produces unnecessary or redundant code, such as assigning values to variables that are not read or used in subsequent parts of the program. This not only increases the complexity of the code but can also affect its execution efficiency. In Figure 9 (b) from MBPP-45, the example of the generated code shows that although the main purpose of the code is to compute the greatest common divisor (GCD) of a list of numbers, the code includes operations for assigning initial values to *num1* and *num2*.

R4-3 Inefficient Exception Handling. The generated code includes improper exception handling, which can become a performance bottleneck in scenarios requiring frequent calls (such as in loops or core processing logic). In Figure 9 (c) from AixBench-72, the generated code contains issues with ex-

```
# Generated code
def len_complex(a,b):
  x=a**2
  y=b**2
  z=x+y
  length=z**0.5
  return length
```

```
# Canonical solution
def len_complex(a,b):
  cn=complex(a,b)
  length=abs(cn)
  return length
```

(a) Underutilization of Language Features from MBPP-688

```
def get_gcd(l):
  num1 = l[0]
  num2 = l[1]
  gcd = num1
  for i in range(2, len(l)):
    num2 = l[i]
    while(num2):
      num1, num2 = num2, num1 % num2
    gcd = num1
  return gcd
```

(b) Unused Variables from MBPP-45

```
public <T> T newInstance(Class<T> clazz) {
  try{
    return clazz.newInstance();
  }catch (InstantiationException|IllegalAccessException e){
    throw new RuntimeException(e);
  }
```

(c) Inefficient Exception Handling from AixBench-72

Fig. 9: Examples of inefficient use of language features.

ception handling, which may lead to performance regression. The code frequently throws and catches specific exceptions such as *InstantiationException* and *IllegalAccessException*, which respectively indicate problems with class instantiation and access. These exceptions are rewrapped and thrown as *RuntimeException*, a practice that obscures the specific cause of the errors, affecting the performance of the code.

Developers should be aware of these common inefficiencies and apply best practices in code review and testing. Furthermore, the root causes can contribute to a more comprehensive understanding of AI-assisted code generation and can inform the development of future tools and best practices for developers.

C. RQ3: Can prompt engineering optimize AI-generated code for performance?

Motivation: Building upon the significant prevalence of performance regressions identified in AI-generated code in RQ1 and RQ2, this research question aims to investigate the potential mitigation strategies. Since modifying the underlying AI model of GitHub Copilot might not be readily feasible, we explore prompt engineering as a promising approach to improve the performance characteristics of the generated code. Our goal is to use prompts to guide Copilot to generate code that is both functionally correct and optimized for performance. By effectively incorporating performance considerations into prompts, developers can potentially improve overall development productivity and the performance of the generated code.

Approach: We follow three steps to evaluate prompt engineering for performance in AI-generated code.

In the first step, we design prompts. This step involves designing two types of prompts, i.e., general prompts and specific prompts based on the root causes of performance regressions identified in RQ2.

General Prompt: *Complete the function with better-performing Python/Java code. Use efficient function calls and looping structures, use efficient algorithms, avoid unnecessary complexity and waste of resources, ensure that the code is concise, and make full use of language features.* The general prompt aims to enhance performance across a wide range of code generation tasks.

Specific Prompts: Based on the performance regression root causes identified in RQ2, we develop four specific prompts.

- **For code with inefficient function calls:** *Be mindful to choose better-performing function interfaces when calling functions, avoiding unnecessary deep recursion.*
- **For code with inefficient looping:** *Avoid string concatenations and assignments within loops, and optimize the code structure to reduce nested loops.*
- **For code with inefficient algorithms:** *Be sure to use efficient algorithms, minimize unnecessary conditional statements, and for mathematical problems, prioritize algorithms that can exploit mathematical properties.*
- **For code with inefficient use of language features:** *Integrating programming language features, using efficient built-in functions of programming language, avoiding poor exception handling, and reducing unused variables.*

In the second step, we generate code for the scripts from the HumanEval, AixBench, and MBPP datasets by applying general prompt engineering. We then conduct static and dynamic analysis on the code generated with general prompts. Next, we compare the performance obtained in this stage with the previous baseline results.

In the final step, we regenerate code using specific prompts. Based on the scripts after the general prompt, we use specific prompts on the scripts with root causes in RQ2. We then perform static and dynamic analysis on the resulting code and analyze the performance of this code against the baseline and general prompt results.

Results: Prompt engineering can effectively improve the performance of Copilot-generated code. Table IV clearly shows that applying general prompts resulted in a reduction in performance regressions across all three datasets, i.e., HumanEval, AixBench, and MBPP. This reduction is further amplified when specific prompts are applied, targeting identified root causes. For example, in the AixBench dataset, the number of performance regressions drops from 38 (baseline approach) to 29 with general prompts and further down to 20 with specific prompts.

Dynamic analysis reinforces the positive impact of prompt engineering. In both HumanEval and MBPP datasets, all three key performance metrics (execution time, memory usage, and CPU utilization) exhibit a decrease in performance regressions after applying general or specific prompts. For instance, the HumanEval dataset saw a reduction in performance regression instances from 54 (baseline approach) to 38 with specific prompts applied. This indicates that prompt engineering can

TABLE IV: The number of performance regression instances before and after general and specific prompt engineering in static and dynamic analyses

Prompt	Static performance regression analysis				Dynamic performance regression analysis					
	HumanEval	MBPP	AixBench		HumanEval			MBPP		
	Qodana	Qodana	Spotbugs	PMD	Execution time	Memory	CPU	Execution time	Memory	CPU
No prompt (baseline)	14	274	8	38	31	14	54	226	14	265
General prompt	10	265	6	29	18	13	45	174	9	233
Specific prompt	9	259	4	20	14	13	38	168	8	212

effectively optimize the performance of Copilot-generated code during runtime.

The most effective specific prompt is for code with inefficient looping. In the experiment with specific prompts, we obverse that specific prompts targeting inefficient looping deliver the most significant performance improvements. This can be attributed to the relatively straightforward nature of these issues and the ease with which code modifications can be guided through specific prompts. This insight suggests that addressing inefficient looping through prompt engineering could be a priority for achieving performance gains in AI-generated code.

Prompt engineering is most successful in optimizing execution time and CPU utilization. Prompt engineering effectively improves execution time and CPU utilization, as evidenced by reduced performance regressions for these metrics. However, the improvements in memory usage are less prominent. We hypothesize that the reasons for this may include the complexity of memory optimization, which often involves intricate data structures and algorithmic changes that are not as directly addressed by our current prompt engineering strategies. Additionally, memory management is highly dependent on the runtime environment and garbage collection mechanisms, which may not be fully captured by our prompt directives. Further research is needed to develop more effective prompts for memory optimization.

Both general and specific prompts contribute to performance improvement of AI-generated code, with specific prompts demonstrating a significant impact on inefficient looping. While execution time and CPU utilization benefit greatly from prompt engineering, memory usage improvements require further research to fully understand and address the underlying reasons.

V. Discussion

Performance regressions of AI-generated code across languages. While AI-generated code can be functionally correct, it often exhibits performance regressions compared to human-written code. Additionally, our study indicates a higher prevalence of performance regressions in Java-based (25% in AixBench) code generated by Copilot compared to Python (8% in HumanEval and 21% in MBPP). The disparity in performance regressions between Java and Python may be attributed to differences in language constructs, compiler optimizations, or the specific challenges each language presents to the AI model. The datasets we utilized are valuable for our study due to their established use in prior research and comprehensive coverage of typical programming tasks. However, they might not fully capture the complete spectrum of performance-critical code. Despite these limitations, our selection of datasets provides a solid foundation for evaluating AI-generated code, particularly in Java and Python, offering a meaningful perspective on the capabilities and limitations of current LLMs in generating efficient code.

The reasons of MBPP dataset has much more performance regressions. We speculate several factors might be at play: (1) The lack of clear prompts in MBPP compared to HumanEval might lead to ambiguities for Copilot. (2) MBPP's complexity, particularly with math-related questions, poses greater challenges for Copilot. The length of generated code in MBPP varies significantly based on the question difficulty. This variability might introduce additional complexity for Copilot's generation process, potentially impacting performance. (3) It's interesting to note that the creators of Codex (the foundation for Copilot), also developed the HumanEval dataset. This potentially means Copilot might have been trained on tasks similar to HumanEval, leading to better performance on this dataset compared to MBPP.

Prompt engineering for the performance of AI-generated code. Despite the potential of specific prompts to mitigate performance regressions, a significant portion of performance regressions remains, suggesting a need for more sophisticated AI models that can better comprehend code performance. This may involve advanced machine learning techniques that can better predict and prioritize optimization opportunities. We designed two types of prompts, i.e., general prompts and specific prompts based on the root causes of performance regressions. In our approach, we utilized one-shot learning, providing a baseline evaluation. Nonetheless, we recognize the potential of more sophisticated prompt strategies like few-shot learning and chain-of-thought prompting to improve the performance of AI-generated code. Future research is warranted to explore the effectiveness of these deeper prompt strategies in mitigating performance regressions in AI-generated code.

Further Research on the performance regression of AI-generated code. We investigate the performance regressions of code generated by GitHub Copilot, based on the GPT model [38], considered representative of code generation LLMs. This suggests that the observed performance regressions might extend to other LLMs utilizing similar architectures. By employing both static and dynamic analysis techniques, we have provided a comprehensive evaluation of performance regressions. This dual approach helps mitigate the limitations of relying solely on one type of analysis, ensuring a more holistic view of code performance. Our findings point to several areas for further research and development, including a more granular analysis of performance regressions in different programming languages, the development of more advanced AI models capable of understanding and generating high-performance code, and the creation of a wider array of prompts to tackle the variability in performance regression root causes.

VI. Threats

External validity. Our findings are based on three open-source datasets, i.e., HumanEval, AixBench, and MBPP. While valuable, these datasets might not fully represent the spectrum of performance-critical code in diverse real-world applications. This study focuses on Java and Python, and results might not generalize directly to other programming languages. Performance regressions may significantly differ due to language features, compiler optimizations, and running environments. While GitHub Copilot, based on the GPT-3 model, represents code generation LLMs, our findings may not fully apply to other LLMs with different architectures or training data. Future work can incorporate diverse datasets, languages, and LLMs to develop a more generalized understanding of AI-generated code performance regressions.

Internal validity. We employ three static analysis tools (e.g., Qodana, Spotbugs) and three dynamic profilers (e.g., cProfile) to detect potential performance regressions. These tools were chosen for their established use in prior research and their robust support for the programming languages. These tools may not identify all performance regressions, potentially leading to an incomplete understanding of some root causes. To mitigate this limitation, we employed a combination of static and dynamic analysis techniques for a more comprehensive assessment of performance regressions. Future research can explore additional techniques or tools to uncover more complex performance problems within AI-generated code. Manually classifying reasons for performance regressions in AI-generated code can introduce subjective factors. To mitigate this, we employ two authors for independent code examination. Disagreements are resolved with a tie-breaker to ensure consistency. We calculate Cohen's Kappa statistic of 0.88, indicating considerable agreement [34]. Further user and case studies could strengthen this area and provide deeper insights into the rationale behind these regressions. Our approach relies on specific performance metrics (e.g., CPU and memory usage) chosen based on the software systems' nature. While these are common choices, selecting appropriate metrics can require system-specific expertise. Future work could explore including more performance metrics tailored to the characteristics of the subject systems.

Construct validity. Dynamic analysis using profilers can be influenced by environmental factors and noise. To mitigate this, we employ a clean environment and execute the generated code multiple times. However, some noise is inherent in performance monitoring. Future studies could consider increasing repetitions based on time and resource constraints.

VII. Related work

AI-assisted code generation. Extensive prior research has explored automated code generation. Existing techniques fall into two main categories: learning-based and retrieval-based approaches. The learning-based approach focuses on extracting natural language features from training data and using them for code generation. It can be further subdivided into supervised learning [32, 58, 42, 26, 52, 49] and pre-trained model approaches [13, 21, 2, 51, 36, 31]. Supervised learning methods often employ sequence-to-sequence models, which follow an encoder-decoder structure. Pre-trained models, on the other hand, leverage self-supervised training on vast amounts of unlabeled data. Notably, the Transformer architecture is prevalent in pre-trained models for code generation. Researchers have developed specialized pre-trained models for the code domain, achieving impressive results in code generation tasks. Given the vast size of the code generation solution space, retrieval-based approaches incorporate similar code retrieval to assist the decoder in generating code [12, 23, 20, 39, 63]. This approach effectively reduces the decoding space, leading to improved quality in the generated code.

Assessing the quality of code generation techniques. Many studies have evaluated the quality of code generation techniques or tools, e.g., ChatGPT, Copilot, and CodeWhisperer, primarily focusing on whether these tools produce code that fulfills its intended function. Studies like Yetistiren et al. [56] highlighted GitHub Copilot's ability to generate valid code with a high success rate. Sobania et al. [46] found no significant difference in correctness between Copilot and other approaches. Similarly, Nguyen and Nadi [35] and Burak et al. [57] evaluate code correctness, efficiency, and overall quality, with Burak et al. observing improvements in generated code over time. However, recent research has begun to emphasize user experience and the broader impact of these tools on developer productivity. Barke et al. [5] showed that while Copilot might not directly shorten development time, it often serves as a valuable starting point, though challenges remain in understanding, editing, and debugging generated code snippets. Sila et al. [30] compared human-written code with AlphaCode-generated code, emphasizing the need for developer review to identify performance bottlenecks. Coignion et al. [9] found that although LLM-generated code performs well in some cases, it is slower than 27% of human-written code on the LeetCode dataset. Liu et al. [33] found three instances of inefficient implementations within the HumanEval ground truth, which caused slow performance on inputs of reasonable size. The prior studies underscore the need for a more comprehensive evaluation methodology that considers not only functional correctness but also potential performance implications. Hou et al. [24] found performance limitations in GPT-4-generated code, which can be partially addressed through optimization. Garg et al. [17] proposed leveraging LLMs with prompt engineering to optimize code performance, showing effective improvements in addressing performance regressions. However, a comprehensive analysis of performance regressions in AI-generated code is still needed to identify root causes and guide further improvements.

Code performance analysis. Extensive research has been conducted to analyze performance at the code level, which is typically divided into two main categories: static code performance regression analysis and dynamic performance regression analysis. Static analysis examines code without execution, identifying potential performance regressions through code structure and patterns, e.g., performance anti-pattern [8, 43, 11]. Many static analysis tools have also been proposed to analyze code performance regression. For instance, Qodana [45], developed by JetBrains, is a comprehensive static analysis engine that supports identifying a wide array of issues, including performance regressions. Other tools like Spotbugs [48] and PMD [41] are tailored for Java, focusing on detecting bugs. Dynamic analysis involves running the code and measuring performance in a real-world environment. This approach provides insights into the actual runtime behavior of the code by executing unit tests [44, 6] and profiling [54, 53].

VIII. Conclusion

In this work, we investigate the performance regressions of code generated by GitHub Copilot, a large language model (LLM) code generation tool. Our findings demonstrate that while Copilot effectively produces functionally correct code, it often falls short in terms of performance compared to human-written solutions. Analysis reveals that common code-level root causes, such as inefficient function calls and inefficient looping, contribute to these performance regressions. Our exploration of prompt engineering suggests its potential as a strategy for mitigating these performance regressions and improving the performance of AI-generated code. Overall, this study highlights the potential of LLMs for code generation, while also emphasizing the need for further development to optimize their output for performance-critical applications.

Acknowledgment

This work is partly supported by the National Natural Science Foundation of China (Grant Nos. 62302347 and 62141221).

References

[1] Python Software Foundation. 2001-2024. *The Python Profilers*. Accessed: 2024-4-25. 2024. URL: https://docs.python.org/3/library/profile.html.

[2] Wasi Uddin Ahmad et al. "Unified Pre-training for Program Understanding and Generation". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*. Association for Computational Linguistics, 2021, pp. 2655–2668.

[3] *Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot*. https://anonymous.4open.science/r/Performance-copilot-9E1D.

[4] Jacob Austin et al. "Program synthesis with large language models". In: *arXiv preprint arXiv:2108.07732* (2021).

[5] Shraddha Barke, Michael B. James, and Nadia Polikarpova. "Grounded Copilot: How Programmers Interact with Code-Generating Models". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 85–111.

[6] Jinfu Chen et al. "IoPV: On Inconsistent Option Performance Variations". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 2023, pp. 845–857.

[7] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[8] Tse-Hsun Chen et al. "Detecting performance anti-patterns for applications developed using object-relational mapping". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 2014, pp. 1001–1012.

[9] Tristan Coignion, Clément Quinton, and Romain Rouvoy. "A Performance Study of LLM-Generated Code on Leetcode". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 79–89.

[10] Zishuo Ding, Jinfu Chen, and Weiyi Shang. "Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet?" In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 1435–1446.

[11] Imara van Dinten et al. "The slow and the furious? Performance antipattern detection in Cyber-Physical Systems". In: *J. Syst. Softw.* 210 (2024), p. 111904.

[12] Dawn Drain et al. "Generating Code with the Help of Retrieved Template Functions and Stack Overflow Answers". In: *CoRR* abs/2104.05310 (2021).

[13] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*. Vol. EMNLP 2020. Findings of ACL. Association for Computational Linguistics, 2020, pp. 1536–1547.

[14] 2024 Python Software Foundation. *memory-profiler 0.61.0*. Accessed: 2024-4-25. 2024. URL: https://pypi.org/project/memory-profiler/.

[15] 2024 Python Software Foundation. *psutil 5.9.8*. Accessed: 2024-4-25. 2024. URL: https://pypi.org/project/psutil/.

[16] Yujia Fu et al. "Security Weaknesses of Copilot Generated Code in GitHub". In: *CoRR* abs/2310.02059 (2023).

[17] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. "Rapgen: An approach for fixing code inefficiencies in zero-shot". In: *arXiv preprint arXiv:2306.17077* (2023).

[18] Inc. GitHub. *About GitHub Copilot Individual*. Accessed: 2024-4-22. 2024. URL: https://docs.github.com/en/copilot/copilot-individual/about-github-copilot-individual.

[19] Inc. GitHub. *Getting started with GitHub Copilot*. Accessed: 2024-4-22. 2024. URL: https://docs.github.com/en/copilot/using-github-copilot/getting-started-with-github-copilot.

[20] Daya Guo et al. "Coupling Retrieval and Meta-Learning for Context-Dependent Semantic Parsing". In: *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. Association for Computational Linguistics, 2019, pp. 855–866.

[21] Daya Guo et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[22] Yiyang Hao et al. "Aixbench: A code generation benchmark dataset". In: *arXiv preprint arXiv:2206.13179* (2022).

[23] Shirley Anugrah Hayati et al. "Retrieval-Based Neural Code Generation". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 2018, pp. 925–930.

[24] Wenpin Hou and Zhicheng Ji. "A systematic evaluation of large language models for generating programming code". In: *arXiv preprint arXiv:2403.00894* (2024).

[25] Yizhan Huang et al. "Do Not Give Away My Secrets: Uncovering the Privacy Issue of Neural Code Completion Tools". In: *CoRR* abs/2309.07639 (2023).

[26] Srinivasan Iyer et al. "Mapping Language to Code in Programmatic Context". In: *Proceedings of the 2018 Conference on Empirical*

Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018. Association for Computational Linguistics, 2018, pp. 1643–1652.

[27] Mostafa Jangali et al. "Automated Generation and Evaluation of JMH Microbenchmark Suites From Unit Tests". In: *IEEE Trans. Software Eng.* 49.4 (2023), pp. 1704–1725.

[28] Raphaël Khoury et al. "How Secure is Code Generated by ChatGPT?" In: *IEEE International Conference on Systems, Man, and Cybernetics, SMC 2023, Honolulu, Oahu, HI, USA, October 1-4, 2023*. IEEE, 2023, pp. 2445–2451.

[29] Christoph Laaber and Philipp Leitner. "An evaluation of open-source software microbenchmark suites for continuous performance assessment". In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 119–130.

[30] Sila Lertbanjongngam et al. "An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode". In: *16th IEEE International Workshop on Software Clones, IWSC 2022, Limassol, Cyprus, October 2, 2022*. IEEE, 2022, pp. 10–15.

[31] Yujia Li et al. "Competition-Level Code Generation with AlphaCode". In: *CoRR* abs/2203.07814 (2022).

[32] Wang Ling et al. "Latent Predictor Networks for Code Generation". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[33] Jiawei Liu et al. "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation". In: *Advances in Neural Information Processing Systems* 36 (2024).

[34] Mary L McHugh. "Interrater reliability: the kappa statistic". In: *Biochemia medica* 22.3 (2012), pp. 276–282.

[35] Nhan Nguyen and Sarah Nadi. "An Empirical Evaluation of GitHub Copilot's Code Suggestions". In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 1–5.

[36] Erik Nijkamp et al. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[37] Liang Niu et al. "CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023, pp. 2133–2150.

[38] OpenAI. *Powering next generation applications with OpenAI Codex*. Accessed: 2024-7-31. 2024. URL: https://openai.com/index/codex-apps/.

[39] Md. Rizwan Parvez et al. "Retrieval Augmented Code Generation and Summarization". In: *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*. Association for Computational Linguistics, 2021, pp. 2719–2734.

[40] Hammond Pearce et al. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 754–768.

[41] 2024 PMD. *PMD:An extensible cross-language static code analyzer*. Accessed: 2024-4-25. 2024. URL: https://pmd.github.io/.

[42] Maxim Rabinovich, Mitchell Stern, and Dan Klein. "Abstract Syntax Networks for Code Generation and Semantic Parsing". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Association for Computational Linguistics, 2017, pp. 1139–1149.

[43] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. "On the Validity of Performance Antipatterns at Code Level". In: *Softwaretechnik-Trends* 39.4 (2019), pp. 32–34.

[44] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. "PeASS: A Tool for Identifying Performance Changes at Code Level". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 1146–1149.

[45] 2000-2024 JetBrains s.r.o. *About Qodana*. Accessed: 2024-4-25. 2024. URL: https://www.jetbrains.com/help/qodana/about-qodana.html.

[46] Dominik Sobania, Martin Briesch, and Franz Rothlauf. "Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming". In: *GECCO '22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*. ACM, 2022, pp. 1019–1027.

[47] 2024 sonarqube. *Sonarqube: a code quality management platform*. Accessed: 2024-4-28. 2024. URL: https://www.sonarqube.org/.

[48] *SpotBugs:Find bugs in Java Programs*. Accessed: 2024-4-25. 2024. URL: https://spotbugs.github.io/.

[49] Zeyu Sun et al. "TreeGen: A Tree-Based Transformer Architecture for Code Generation". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 8984–8991.

[50] *Visual Studio Code - Code Editing*. https://code.visualstudio.com/.

[51] Yue Wang et al. "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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[52] Bolin Wei et al. "Code Generation as a Dual Task of Code Summarization". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 2019, pp. 6559–6569.

[53] Lingmei Weng et al. "Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling". In: *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 2023, pp. 1–17.

[54] Dacong Yan, Guoqing Xu, and Atanas Rountev. "Uncovering performance problems in Java applications with reference propagation profiling". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 134–144.

[55] Zhou Yang et al. "Gotcha! This Model Uses My Code! Evaluating Membership Leakage Risks in Code Models". In: *CoRR* abs/2310.01166 (2023).

[56] Burak Yetistiren, Isik Ozsoy, and Eray 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, PROMISE 2022, Singapore, Singapore, 17 November 2022*. ACM, 2022, pp. 62–71.

[57] Burak Yetistiren et al. "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT". In: *CoRR* abs/2304.10778 (2023).

[58] Pengcheng Yin and Graham Neubig. "A Syntactic Neural Model for General-Purpose Code Generation". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Association for Computational Linguistics, 2017, pp. 440–450.

[59] Joy He-Yueya et al. "Solving math word problems by combining language models with symbolic solvers". In: *arXiv preprint arXiv:2304.09102* (2023).

[60] Daoguang Zan et al. "Large language models meet nl2code: A survey". In: *arXiv preprint arXiv:2212.09420* (2022).

[61] Yi Zeng et al. "Studying the characteristics of logging practices in mobile apps: a case study on F-Droid". In: *Empirical Software Engineering* 24.6 (2019), pp. 3394–3434.

[62] Zibin Zheng et al. "A survey of large language models for code: Evolution, benchmarking, and future trends". In: *arXiv preprint arXiv:2311.10372* (2023).

[63] Shuyan Zhou et al. "DocPrompting: Generating Code by Retrieving the Docs". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.