Empirical Software Engineering (2026) 31:62
https://doi.org/10.1007/s10664-025-10776-1

RESEARCH PAPERS

Performance analysis of AI-generated code: A case study of Copilot, Copilot Chat, CodeLlaMa, and DeepSeek-Coder models

Shuang Li[1] · Yuntao Cheng[1] · Jinfu Chen[1,2] · Jifeng Xuan[1] · Sen He[3] · Weiyi Shang[4]

Received: 5 March 2025 / Accepted: 11 November 2025

Abstract
The integration of Large Language Models (LLMs) into software development tools like GitHub Copilot and Copilot Chat, along with the advancement of code generation models like DeepSeek-Coder and CodeLlama, hold the promise of transforming code generation processes. While AI-driven code generation presents numerous advantages for software development, code generated by LLMs 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, Copilot Chat, CodeLlama, and Deepseek-Coder across four distinct datasets: HumanEval, AixBench, MBPP, and the performance-oriented benchmark EvalPerf. 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 algorithms, and inefficient use of language features. We further identify a total of eleven sub-categories of root causes attributed to the performance regressions of generated code. Additionally, we explore prompt engineering including few-shot and Chain-of-Thought (CoT) prompting as a potential strategy for optimizing performance. The outcomes demonstrate that few-shot prompting, grounded in identified root causes of code performance regressions, can improve the performance of generated code by guiding models toward performance-oriented generation. In contrast, CoT prompting proves less effective, and in some cases detrimental, suggesting that reasoning-oriented strategies do not necessarily enhance performance. Across both general-purpose and efficiency-oriented benchmarks, our analysis reveals that performance regressions persist regardless of dataset scope, underscoring the necessity of treating performance as a first-class dimension of code quality. This research provides valuable insights that contribute to a more comprehensive understanding of AI-assisted code generation.

Communicated by: Lei Ma, Roberto Pietrantuono.

Extended author information available on the last page of the article

Published online: 19 January 2026

Keywords Code Generation · Software Performance · Program Analysis · Performance Engineering · Large Language Models

1 Introduction

The integration of LLMs into software development tools has introduced a new era of AI-powered coding assistants. These LLM-based tools, such as GitHub Copilot, Copilot Chat, ChatGPT, and CodeWhisperer, as well as code generation models like CodeLlama and DeepSeek-Coder, 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. Its extension, GitHub Copilot Chat, provides an interactive conversational interface that enables developers to request explanations and debugging assistance, thereby further integrating LLMs into the software development workflow. The LLM-based code generation tools offer the potential to enhance developer productivity and streamline development processes. Similarly, emerging models like CodeLlama and DeepSeek-Coder further expand the applicability of LLMs in code generation, significantly influencing the methodologies and workflows of software engineers.

While tools like Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder offer 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 (Nguyen and Nadi 2022; Yetistiren et al. 2022, 2023; Wang et al. 2025; Mayer et al. 2024), security (Pearce et al. 2022; Fu et al. 2025; Khoury et al. 2023; Zhang et al. 2024), and privacy (Niu et al. 2023; Huang et al. 2023; Yang et al. 2023; Luo et al. 2024a) 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, particularly its variability across different code generation models, 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. Code efficiency 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 mainstream code generation models such as GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder and evaluating the performance regressions comprehensively using both static analysis tools and dynamic profiling. Through this approach, we aim to quantify the performance regression differences among various models in the code generation process. To ensure the broad applicability of our findings, we select four diverse and representative datasets, i.e., HumanEval, AixBench, MBPP, and the performance-oriented benchmark EvalPerf. 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, tracemalloc, 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 causes that contribute to these regressions, including inefficient function calls, inefficient looping, inefficient algorithm, and inefficient use of language features. To address these performance regressions, we explore prompt engineering as a mitigation strategy. In particular, we propose a few-shot prompting approach that incorporates concrete examples reflecting the identified root causes, guiding models toward generating more efficient code. We also evaluate CoT prompting to examine whether reasoning-oriented strategies can enhance performance optimization. The results show that few-shot prompting could mitigate the performance regressions of code generation models, whereas CoT prompting yields limited or inconsistent benefits, suggesting that explicit reasoning does not directly translate into efficiency gains.

To facilitate research reproducibility, we make the original datasets, scripts, profiling results, and analysis rules available in our replication package[1]. Our contributions are summarized below.

- **Performance assessment of AI-generated code:** We systematically evaluated the performance regressions between code generated by GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder and canonical solutions. Using static analysis tools and dynamic profiling, we analyzed the performance regressions of these code generation models across different tasks and datasets including HumanEval, MBPP, AixBench, and performance-oriented benchmark EvalPerf. The study revealed that performance regression is a common phenomenon in AI-generated code. Compared to human-written code, the generated code exhibits significant inefficiencies in terms of execution efficiency and resource utilization with the proportion of performance regressions being even more pronounced on the performance-oriented benchmark EvalPerf.
- **Performance regression root causes of AI-generated code:** We qualitatively analyzed the root causes of performance regression in the generated code. Four major categories of inefficiencies were identified: inefficient function calls, inefficient loops, inefficient algorithms, and suboptimal use of language features. These were further refined into eleven specific subcategories. We conducted this root cause analysis for all models across all datasets, highlighting how different models exhibit distinct patterns of performance regressions depending on the dataset. These findings offer valuable insights into the potential shortcomings of current AI-driven coding assistants and establish a solid foundation for devising strategies to optimize the performance of AI-generated code.
- **Prompt engineering for performance optimization:** We explored prompt engineering including few-shot and CoT prompting as a technique for optimizing the performance of AI-generated code and proposed a few-shot prompt engineering optimization strategy. By embedding specific root causes from the identified subcategories into the few-shot prompt design, we enhanced the performance of the generated code in most cases, although the improvement was limited or inconsistent for certain models and datasets. In contrast, CoT prompting, while reasoning-oriented, does not consistently improve performance and can sometimes degrade performance. These findings suggest that the

[1] https://github.com/hebena/Performance-Analysis-of-AI-Generated-Code

effectiveness of prompt engineering is highly model- and dataset-dependent. This highlights the importance of aligning prompt design with concrete performance optimization objectives and provides actionable insights for future work on efficiency-aware code generation with LLMs.

This work systematically extends our previous work Li et al. (2024b). First, we add more AI-powered coding assistants, i.e., Copilot Chat, CodeLlama, and DeepSeek-Coder, to more comprehensively evaluate the performance of AI-generated code and investigate the performance discrepancy among different models in code generation tasks. Second, we extend our evaluation with a newly introduced performance-oriented dataset, EvalPerf, which provides a richer set of benchmarks for assessing performance regressions in AI-generated code. Third, we investigate prompt engineering strategies, including few-shot and CoT prompting, to study their impact on performance regressions across different models and datasets. In addition, we incorporated detailed profiling analyses and manual inspections of static analysis results to uncover dominant sources of performance regressions, and included statistical testing to ensure the robustness of observed effects. Finally, we provide a discussion of model-specific performance differences, architectural implications, and prompt sensitivity, offering insights into how design choices, model characteristics, and prompt strategies jointly shape the performance of AI-generated code, providing actionable guidance for future research on performance-aware AI-assisted code generation.

The rest of this paper is organized as follows. Section 2 introduces how to generate code using GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder and a motivation example. Section 3 presents our case study setup. Section 4 presents detailed results of our research questions and findings. Section 5 discusses the implications of our findings. Section 6 discusses the threats to the validity of our study. Section 7 presents related prior studies. Section 8 summarizes the conclusion of our study.

2 Background and Motivating Example

In this section, we first introduce four AI-powered code generation models, GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder. We then present a motivating example of performance challenges in AI-generated code.

2.1 AI-Powered Code Generation Models

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.

GitHub Copilot Chat builds upon GitHub Copilot and represents a more advanced evolution of AI-powered coding assistants. Unlike Copilot's inline code completion, Copilot Chat provides an interactive conversational interface that leverages AI models to support a wide range of developer interactions. This interface enables developers not only to request code completions, but also to ask for explanations, debugging guidance, and refactoring suggestions (GitHub Docs 2025a).

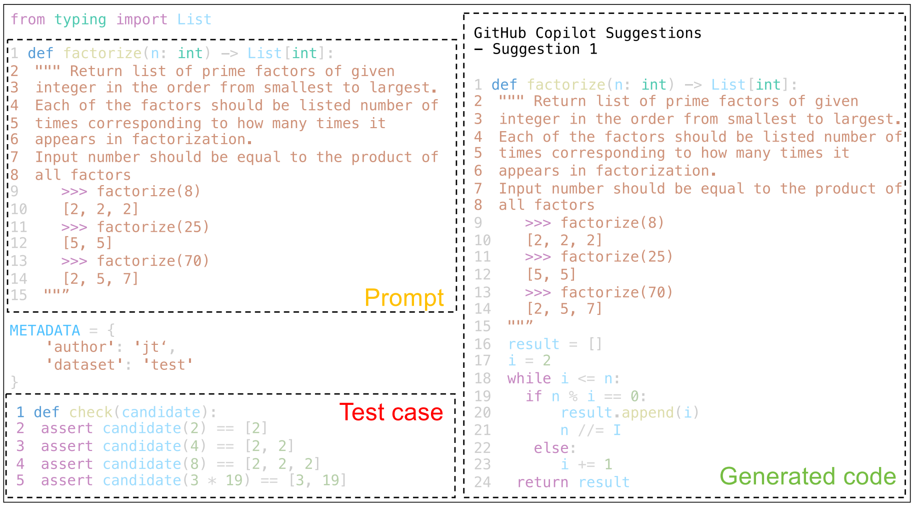
In addition to Copilot Chat, other AI-powered code generation models, such as CodeLlama and DeepSeek-Coder have emerged in recent years. These models incorporate more advanced language understanding capabilities and diverse architectural designs, achieving continuous breakthroughs in code generation accuracy and efficiency. For instance:

- CodeLlama, built on the Llama 2 architecture, specializes in code generation and offers features such as infilling capabilities, support for large input contexts, and zero-shot instruction-following for programming tasks. It supports multiple programming languages, including Python, C++, Java, PHP, TypeScript, C#, and Bash (Rozière et al. 2023).
- DeepSeek-Coder provides a range of models scaling from 1B to 33B parameters, focusing on code and natural language tasks. Its features include code completion, code insertion, and repository-level code suggestions (Guo et al. 2024).

This study evaluates the performance and optimization potential of the code generated by GitHub Copilot , Copilot Chat, CodeLlama, and DeepSeek-Coder. These tools support various popular programming languages, including Python, Java, TypeScript, and C# (GitHub Docs 2025b; Rozière et al. 2023; Guo et al. 2024). In this study, we leverage four models' capabilities to generate code for datasets encompassing two specific languages: Python and Java, including both general-purpose benchmarks and the performance-oriented benchmark. By generating code across these datasets and languages, we conducted a detailed analysis of the performance impact in a language-specific context, providing insights into how model- and language-specific characteristics affect code performance and optimization opportunities.

2.2 A Motivating Example of Performance Challenges in AI-Generated Code

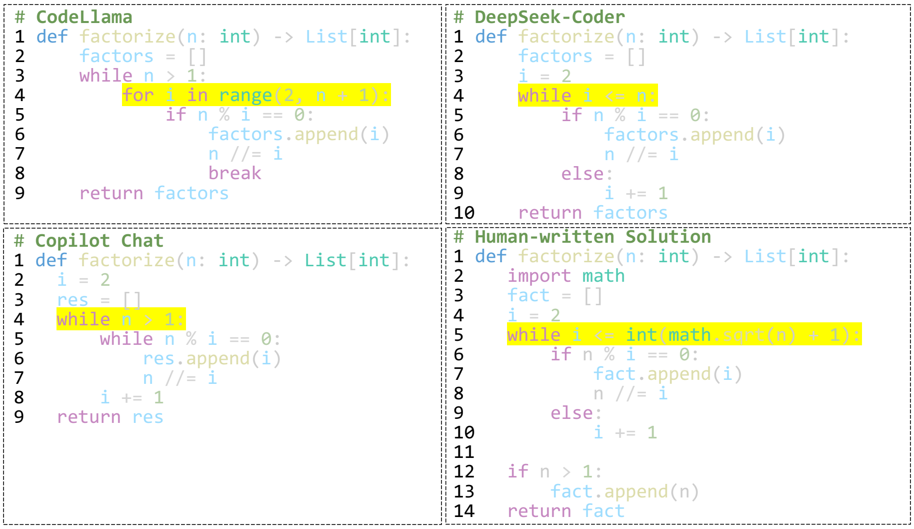
While AI-powered code generation models offer significant potential for developers, a critical aspect to consider is the performance of the generated code. Here, we present a motivating example highlighting this challenge. One wants to develop a function to determine if a given integer is a prime number. To achieve this, one uses GitHub Copilot within the Visual Studio Code (GitHub, Inc. 2025) environment to generate the function. The generated code, while functionally correct, exhibits performance regressions. As illustrated in Fig. 1, the generated code employs a naive approach that iterates from 2 to the original number, leading to inefficiency. This issue is not exclusive to Copilot. As shown in Fig. 2, CodeLlama, for instance, generates code that iterates from 2 to the given number plus one.



```
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 by Copilot



```
# CodeLlama
def factorize(n: int) -> List[int]:
    factors = []
    while n > 1:
        for i in range(2, n + 1):
            if n % i == 0:
                factors.append(i)
                n //= i
                break
    return factors
```

```
# DeepSeek-Coder
def factorize(n: int) -> List[int]:
    factors = []
    i = 2
    while i <= n:
        if n % i == 0:
            factors.append(i)
            n //= i
        else:
            i += 1
    return factors
```

```
# Copilot Chat
def factorize(n: int) -> List[int]:
    i = 2
    res = []
    while n > 1:
        while n % i == 0:
            res.append(i)
            n //= i
        i += 1
    return res
```

```
# Human-written Solution
def factorize(n: int) -> List[int]:
    import math
    fact = []
    i = 2
    while i <= int(math.sqrt(n) + 1):
        if n % i == 0:
            fact.append(i)
            n //= i
        else:
            i += 1

    if n > 1:
        fact.append(n)
    return fact
```

Fig. 2 The generated code for the function to determine all prime factors of a given number from HumanEval-25.py by CodeLlama, DeepSeek-Coder, and Copilot Chat, as well as the human-written solution

Similarly, both DeepSeek-Coder and Copilot Chat adopt a comparable strategy, repeatedly dividing the number while incrementing the divisor, effectively iterating up to the original input size. It is also worth noting the difference between Copilot and Copilot Chat. While both ultimately produce inefficient factorization routines, Copilot's implementation simply loops through possible divisors up to the original number, whereas Copilot Chat nests an additional inner loop to repeatedly divide by the same factor before incrementing. Although functionally correct, this nested structure introduces redundant checks and can further exacerbate performance regressions. This highlights not only the tendency of AI-generated solutions to overlook efficiency but also the variability in generated implementations, which may differ structurally while sharing the same fundamental inefficiency. In contrast, a manually written solution optimizes performance by iterating only to the square root of the target number, significantly reducing computational overhead and improving efficiency. This example emphasizes the potential for AI-generated code to introduce performance regressions. While these tools and models are capable of producing functionally correct code, they do not consistently prioritize the most efficient implementation. Our study aims to bridge this gap by analyzing the performance characteristics of code generated by GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder, and exploring techniques such as few-shot prompt engineering to optimize the efficiency and resource utilization of AI-generated code. By understanding these characteristics, we can develop best practices and techniques to optimize performance and unlock the full potential of AI-powered coding assistants and models.

3 Case Study Setup

In this section, we present our case study setup. In particular, we present the datasets used in our case study and our experimental setup to collect code generated by Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder, and performance regression analysis data.

3.1 Dataset

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

- **HumanEval:** This handcrafted Python dataset by Chen et al. (2021) consists of 164 problems. 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 (Austin et al. 2021). Each problem is presented with a brief description and corresponding test cases.
- **Aixbench:** Designed for Java code generation, Aixbench (Hao et al. 2022) offers 187 problems, along with function signatures and test cases.
- **EvalPerf:** EvalPerf (Liu et al. 2024a) is a Python dataset with 118 tasks focused on execution efficiency. It uses computation-intensive inputs and a relative scoring mechanism to distinguish solutions by performance rather than correctness.

The overview of datasets is shown in Table 1. HumanEval and MBPP datasets are chosen for their broad acceptance within the research community, as highlighted by Zheng et al. (2023) and Zan et al. (2022). These datasets are widely recognized for their robustness and

Table 1 Overview of datasets used in our study

Dataset	Language	#Instances	Year	Reference
HumanEval	Python	164	2021	Chen et al. (2021)
AixBench	Java	187	2022	Hao et al. (2022)
MBPP	Python	974	2021	Austin et al. (2021)
EvalPerf	Python	118	2024	Liu et al. (2024a)

relevance in code generation. The canonical solutions are authored by senior developers, representing high-quality and efficient solutions for specific problems, as documented by prior studies (Chen et al. 2021; Austin et al. 2021).

In addition, we include the EvalPerf dataset, which focuses specifically on performance-oriented programming tasks. EvalPerf contains 118 carefully designed challenges, each with high-computation test inputs, aimed at assessing the efficiency of generated code rather than merely its correctness. By leveraging the Differential Performance Evaluation framework, EvalPerf compares new solutions against a set of reference solutions with varying efficiency levels using a cluster-based scoring mechanism, allowing for finer-grained and stable evaluation of code performance across different tasks and platforms. This makes EvalPerf particularly suitable for benchmarking AI-generated code in terms of execution efficiency. By including EvalPerf alongside general-purpose benchmarks, we ensure that our study covers both functional correctness and performance-critical aspects of AI-generated code.

3.2 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 3 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 four datasets, i.e., HumanEval, AixBench, MBPP, and Evalperf. In the second step, for each prepared prompt, we feed the prepared prompt to GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder 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.

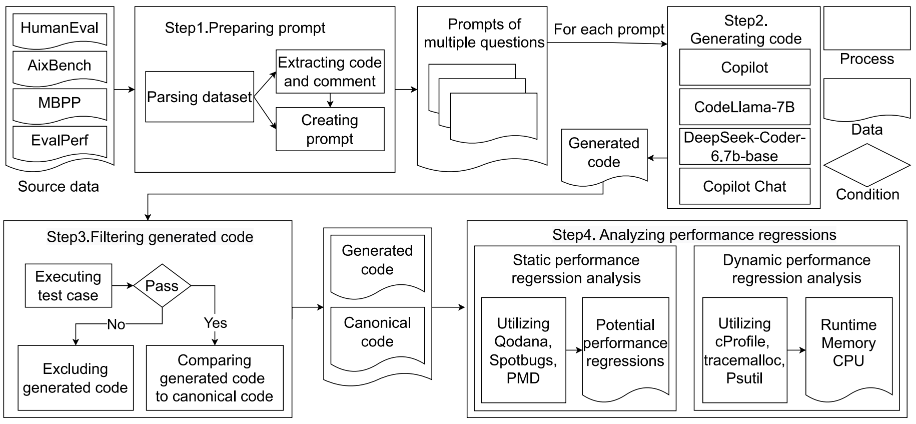
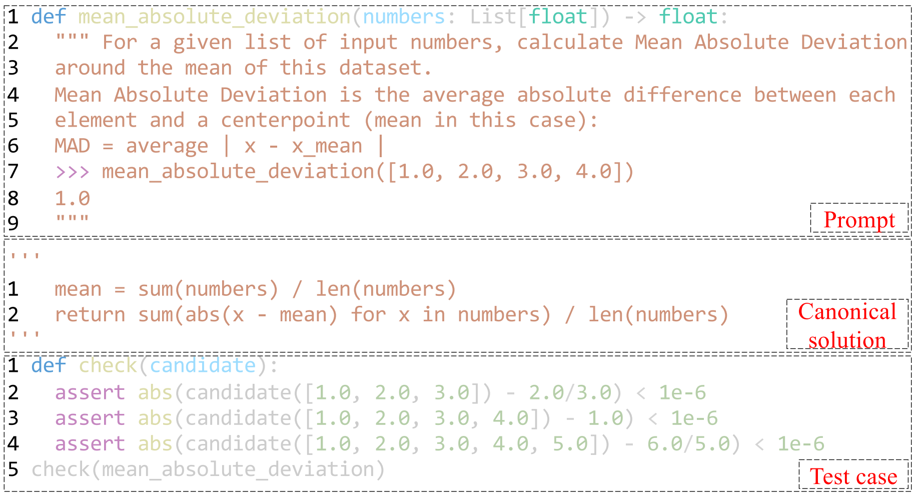

HumanEval
AixBench
MBPP
EvalPerf
Source data
Step1.Preparing prompt
Parsing dataset
Extracting code and comment
Creating prompt
Prompts of multiple questions
For each prompt
Step2. Generating code
Copilot
CodeLlama-7B
DeepSeek-Coder-6.7b-base
Copilot Chat
Generated code
Process
Data
Condition
Step3.Filtering 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, tracemalloc, Psutil
Runtime Memory CPU

Fig. 3 An overview of our approach to collecting data

Step 1: Preparing prompt. The data from HumanEval, MBPP, Evalperf, 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 Fig. 4, 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, Copilot Chat, CodeLlama, and DeepSeek-Coder.

Step 2: Generating code. In this step, we use code generation models to generate code for each prompt of each question. We designed specific parameter configurations for different code generation models and datasets to ensure stability and efficiency during the generation process.

- **GitHub Copilot:** We leverage Copilot within the VSCode (Microsoft 2025) environment to generate code for each prompt. For each newly created file, we open the command panel using *Ctrl+Shift+P* and execute *GitHub Copilot: Open Completions Panel* to activate the suggestions panel. Copilot provides 1-10 code completion suggestions. We consistently accept the first suggestion (see Fig. 1) for consistency. GitHub Copilot was used with its default settings, without any modification of configurable parameters (Visual Studio Code Docs 2025)
- **CodeLlama:** We selected the CodeLlama-7B model, reflecting a balance between model capacity and computational overhead: although larger models generally offer higher accuracy, they also demand substantially more computing resources. Conversely, smaller models often exhibit diminished performance on code generation benchmarks and may have more performance regressions. We set the temperature to "0.0" and "top_p" to "0.9" to balance randomness and determinism in the output. Additionally, we configured "max_gen_len=1024" and "max_batch_size=4" to enable the model to generate sufficiently long code while supporting efficient batch processing. To accommodate the varying prompt lengths across the HumanEval, MBPP, and AixBench datasets, we adjusted the "max_seq_len" parameter to 2048, 5120, and 512, respectively.



```python
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
    """
```

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

```python
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. 4 Examples of prompt preparation, corresponding canonical solution, and the test case from the HumanEval dataset

Since EvalPerf is derived from HumanEval and MBPP, we use the same parameters as MBPP to ensure consistency in evaluation. This ensured that the model could operate properly and effectively utilize contextual information.

– **DeepSeek-Coder:** We selected the deepseek-coder-6.7b-base model, whose parameter scale closely approximates that of CodeLlama-7B. This choice ensures a fair comparison by minimizing performance regressions arising from differences in model size. Additionally, we set "max_new_tokens" to 1024, allowing the model to adapt to prompts of different lengths. This choice was based on experimental observations, ensuring that the generated code was of adequate length to meet the requirements of programming tasks in the selected datasets, while maintaining fluency and consistency when processing longer prompts.
– **Copilot Chat:** We utilize the Copilot Chat feature in VSCode, which directly interacts with the Copilot Chat agent (default GPT-4.1). Input files are passed to Copilot Chat to generate code suggestions. Since Copilot Chat is a production-level tool with advanced internal optimizations, we retain the default settings without manual tuning. This ensures a realistic and practical evaluation of its code generation capabilities across datasets.

Across all dataset experiments, we followed a consistent generation procedure for all files requiring completion within the same model and dataset. The generated files were sequentially named, such as HumanEval-0 through -163, MBPP-1 through -974, and AixBench-0 through -186. For EvalPerf, we retained its original task identifier format, which follows `HumanEval_#` and `Mbpp_#`, to ensure consistency with the dataset design.

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. The correctness rates of this filtering process are shown in Table 2.

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.

Table 2 The correctness rates of Copilot, CodeLlama, DeepSeek-Coder, and Copilot Chat

Dataset	#Instances	Copilot	CodeLlama	DeepSeek-Coder	Copilot Chat
HumanEval	164	134 (81.7%)	110 (67.1%)	84 (51.2%)	151 (92.1%)
AixBench	175	88 (50.3%)	71 (40.6%)	66 (37.7%)	114 (65.1%)
MBPP	974	856 (87.9%)	723 (74.2%)	815 (83.7%)	828 (85.0%)
EvalPerf	118	103 (87.3%)	97 (82.2%)	91 (77.1%)	109 (92.4%)

3.2.1 Static Performance Regression Analysis

We use three static analysis tools: Qodana (JetBrains s.r.o 2024), SpotBugs (2024), and PMD (2024) 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 the 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 (2024). 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" (PMD 2024). The complete definitions and configurations of these custom rules are publicly available in our replication package.

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 generated HumanEval, MBPP, and EvalPerf datasets to projects. In this way, Qodana can automatically identify and highlight potential performance regression issues.

3.2.2 Dynamic Performance Regression Analysis

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

cProfile (Python Software Foundation 2025b): A Python library for performance analysis, providing metrics like the number of function calls and time spent in functions.

tracemalloc (Python Software Foundation 2025c): A module for tracing memory blocks allocated by Python, providing detailed statistics on memory allocation, including tracebacks, memory usage per file/line, and the ability to detect memory leaks by comparing snapshots.

Psutil (Python Software Foundation 2025a): 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. The hardware configuration for the experiment includes an Intel Core i9-13900K processor, 128 GB of RAM, 1 TB SSD for primary storage, and 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.

4 Case Study Results

In this section, we present our case study results by answering three research questions. For each research question, we show the motivation, approach, and corresponding results of the research question.

4.1 RQ1: How Prevalent are Performance Regressions in AI-Generated Code?

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 and code generation models.

Approach Our approach involves two strategies to evaluate the performance of AI-generated code. 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, MBPP, and EvalPerf 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. For the EvalPerf dataset, no explicit canonical solution is provided. Instead, each problem in EvalPerf is associated with multiple reference implementations, each annotated with an efficiency score. Since a higher efficiency score indicates better performance (Liu et al. 2024a), we select the reference implementation with the highest efficiency score as the canonical solution for that problem. This enables us to conduct performance regression analysis in a manner consistent with the HumanEval and MBPP datasets. Using the dynamic performance regression detection modules, we conduct dynamic performance regression analysis on the generated and canonical code sets of these three datasets. 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 (Laaber and Leitner 2018; Ding et al. 2020; Jangali et al. 2023). 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 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 5 illustrates this modification for the script shown in Fig. 4. Once the iterations have 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. For the EvalPerf dataset, we do not perform iterative repetition, since its test cases are already performance-oriented and comparatively larger, making additional iterations unnecessary. Instead, each script is directly profiled in its original form.

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. Memory usage is monitored via the tracemalloc module, which reports both the current memory consumption at the end of execution and the peak memory consumption during execution. CPU utilization is measured using psutil in combination with cProfile, converting runtime statistics into CPU usage ratios to provide a system-level view of resource utilization (Tanenbaum 2015; Psutil Documentation Team 2025).

Results Performance regressions are not rare instances in code generated by Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder.

For Java code, static analysis using SpotBugs and PMD revealed notable performance regressions across all four models. Specifically, Copilot-generated code contained 8 low-performance instances identified by SpotBugs and 41 detected by PMD. In comparison, CodeLlama produced a similar number of regressions, with 8 identified by SpotBugs and 46 by PMD, while DeepSeek-Coder exhibited relatively fewer regressions, with 5 identified by SpotBugs and 22 by PMD. Copilot Chat showed the highest number of flagged issues, with 6 low-performance cases from SpotBugs and up to 74 cases from PMD, indicating more severe static performance concerns compared to the other models. These findings align with

```
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. 5 An example of increasing the number of iterations within the test case of Fig. 4

the differences in functional pass rates on the AixBench dataset, where GitHub Copilot achieved 50.3%, CodeLlama 40.6%, DeepSeek-Coder 37.7%, and Copilot Chat 65.1%. Although DeepSeek-Coder produced the lowest pass rate, its static performance regression counts were lower, suggesting relatively better structural efficiency despite weaker functional correctness. Conversely, Copilot Chat exhibited the best functional correctness on AixBench but suffered from the highest number of performance regressions, highlighting a trade-off between functional accuracy and performance efficiency.

For Python code, static analysis using Qodana identified substantial performance regressions across the HumanEval, MBPP, and Evalperf datasets. Copilot-generated code had 14, 274 and 26 instances of potential performance regressions in these datasets, respectively. CodeLlama performed slightly better, with 8, 188 and 17 instances identified. DeepSeek-Coder exhibited the lowest number of issues in the HumanEval dataset (5 instances) but showed significantly more in the MBPP(262) and EvalPerf (23) datasets. Copilot Chat reported 7 regressions on HumanEval, 66 on MBPP, and 3 on EvalPerf. When considering both regressions and the number of passed instances, Copilot Chat consistently achieved the most favorable balance. Specifically, it reached the higher pass counts (151 on HumanEval, 828 on MBPP, and 109 on EvalPerf), which corresponded to the lowest regression rates of 4.6%, 8.0%, and 2.8%, respectively. In comparison, GitHub Copilot and DeepSeek-Coder showed high regression rates on MBPP (32.0% and 32.1%) and EvalPerf (25.2% and 25.3%). CodeLlama achieved moderate regression rates (7.3% on HumanEval, 26.0% on MBPP, and 17.5% on EvalPerf). These variations highlight differences in how AI models handle performance optimization across datasets. While GitHub Copilot demonstrated broad applicability in functional implementation, it showed relatively more instances of performance regressions in Qodana's static analysis. CodeLlama exhibited better performance in Qodana's analysis, although its code accuracy was occasionally lower than that of Copilot. DeepSeek-Coder's performance in detecting instances of performance regressions fell between the two, with slightly better results than Copilot on the HumanEval dataset. By contrast, Copilot Chat achieved the most favorable balance in Qodana's static analysis, combining the highest pass counts with the lowest potential regression rates across all datasets.

Dynamic analysis further confirmed that AI-generated code often lags behind human-written solutions. To determine whether observed performance differences constitute meaningful regressions, we adopt a statistical testing approach similar to prior work (Chen et al. 2022). For each generated code and its canonical code, we execute each script 11 times, discard the first run to eliminate warm-up effects, and use the results from the remaining 10 runs for statistical analysis. Specifically, we compare performance metrics using the Mann-Whitney U test to identify statistically significant differences (p-value < 0.05). To complement significance testing, we calculate Cliff's δ to quantify effect sizes. Only differences with medium or large effect sizes ($0.33 <$ Cliff's $\delta \leq 0.474$ for medium, $\delta > 0.474$ for large) are considered indicative of performance regression. This approach ensures that our regression analysis accounts for both statistical significance and practical impact, mitigating misleading conclusions that could arise from trivial differences or small sample sizes.

To further strengthen the causal interpretation of detected performance regressions and mitigate confounding effects arising from code structure differences (e.g., imports, logging, or unused statements), we employ a two-stage profiling design. In the first stage, we perform standard dynamic profiling on all generated and canonical implementations across the three

Python datasets to collect runtime, memory, and CPU utilization statistics. This stage identifies cases exhibiting statistically significant performance regressions based on the aforementioned tests. In the second stage, for those regression cases only, we conduct refined profiling focused exclusively on the critical execution path of the generated implementation. Instead of profiling the entire script–which may include initialization or I/O overhead–we isolate the execution of the core verification function and measure function-level runtime contributions. This refined profiling provides a fine-grained view of which internal functions dominate execution time and clarifies whether the observed regressions stem from algorithmic inefficiencies rather than incidental structural variations.

In the HumanEval dataset, among 134 functionally correct scripts generated by Copilot, 38 showed significant runtime discrepancies. Additionally, 10 scripts exhibited notable memory usage discrepancies, and 56 scripts displayed substantial CPU utilization gaps. Overall, 77 scripts demonstrated significant regression in at least one performance metric compared to the human-written code. The MBPP dataset presents a different set of disparities. 142, 68, and 333 scripts contain performance regression in runtime, memory usage, and CPU utilization, respectively. The EvalPerf dataset shows more inefficiencies. 74, 36, and 51 scripts exhibited significant performance regressions in execution time, memory usage, and CPU utilization, respectively. Table 3 presents the performance regressions observed in Copilot-generated code across these datasets, showing the number of notable performance regressions among scripts that passed the test cases.

Compared to Copilot, CodeLlama exhibited slightly fewer performance regressions in HumanEval and MBPP, but more regressions in EvalPerf. Among 110 functionally correct scripts in HumanEval, 14 had significant runtime differences, 2 displayed memory inefficiencies, and 31 showed CPU utilization regressions. In total, 43 scripts (39%) demonstrated substantial regressions in at least one metric. Similar trends were observed in MBPP, where 113 scripts had runtime regressions, 51 had memory inefficiencies, and 215 exhibited CPU utilization regressions. In the EvalPerf dataset, among 97 functionally correct scripts, 72, 43, and 49 scripts showed regressions in execution time, memory usage, and CPU utilization, respectively. Although Copilot has more functionally correct scripts (103) and thus a higher absolute number of regressions, the regression rates of Codellama are higher, indicating worse overall performance. DeepSeek-Coder displayed further regressions in the HumanEval dataset. Among 84 functionally correct scripts in HumanEval, 27 had significant runtime differences, 8 displayed memory inefficiencies, and 29 showed excessive CPU utilization. 49 scripts exhibited regressions in at least one metric. Its performance in the MBPP dataset was slightly better compared to Copilot and CodeLlama, with 115, 42, and 286 scripts demonstrating regressions in runtime, memory usage, and CPU utilization, respectively. However, on the EvalPerf dataset, its performance was worse than both Copilot and CodeLlama, with 70, 35, and 47 scripts showing regressions in execution time, memory usage, and CPU utilization, respectively.

Copilot Chat exhibited the strongest static performance across all datasets, achieving the highest pass rates (92.1% on HumanEval, 85.0% on MBPP, and 92.4% on EvalPerf) and the lowest Qodana regression counts (7, 66, and 3, respectively). However, dynamic profiling revealed the opposite trend. On HumanEval, 50 scripts showed execution time regressions, 45 memory inefficiencies, and 73 CPU utilization regressions. The disparities became more pronounced in MBPP, with 393 scripts affected by execution time regressions, 275 by memory usage, and 355 by CPU utilization. EvalPerf also showed ineffi-

Table 3 Comparison of static and dynamic performance regression analysis for different models: GitHub Copilot, CodeLlama, DeepSeek-Coder, and Copilot Chat

			Static performance regression analysis			Dynamic performance regression analysis		
Model	Dataset	#Passed instances (%)	SpotBugs	PMD	Qodana	Execution time	Memory usage	CPU Utilization
	HumanEval	134 (81.7%)	N/A	N/A	14	38	10	56
GitHub Copilot	MBPP	856 (87.9%)	N/A	N/A	274	142	68	333
	EvalPerf	103 (87.3%)	N/A	N/A	26	74	36	51
	AixBench	88 (50.3%)	8	41	N/A	N/A	N/A	N/A
	HumanEval	110 (67.1%)	N/A	N/A	8	14	2	31
CodeLlama	MBPP	723 (74.2%)	N/A	N/A	188	113	51	215
	EvalPerf	97 (82.2%)	N/A	N/A	17	72	43	49
	AixBench	71 (40.6%)	8	46	N/A	N/A	N/A	N/A
	HumanEval	84 (51.2%)	N/A	N/A	5	27	8	29
DeepSeek-Coder	MBPP	815 (83.7%)	N/A	N/A	262	115	42	286
	EvalPerf	91 (77.1%)	N/A	N/A	23	70	35	47
	AixBench	66 (37.7%)	5	22	N/A	N/A	N/A	N/A
	HumanEval	151 (92.1%)	N/A	N/A	7	50	45	73
Copilot Chat	MBPP	828 (85.0%)	N/A	N/A	66	393	275	355
	EvalPerf	109 (92.4%)	N/A	N/A	3	82	39	62
	AixBench	114 (65.1%)	6	74	N/A	N/A	N/A	N/A

ciencies, with 82, 39, and 62 regressions in execution time, memory, and CPU utilization, respectively. These results suggest that while Copilot Chat tends to generate functionally correct and statically robust code, it is more prone to runtime inefficiencies compared to other models. The dynamic performance analysis of Copilot, CodeLlama, DeepSeek-Coder, and Copilot Chat consistently revealed notable performance regressions, with varying degrees of severity depending on the dataset and performance metric. Overall, Copilot, CodeLlama, and DeepSeek-Coder exhibited moderate levels of performance regressions on HumanEval and MBPP. In contrast, Copilot Chat showed significantly higher inefficiencies across all dynamic metrics. EvalPerf proved to be the most demanding dataset overall, with all four models exhibiting higher regression rates, and Copilot

Chat again ranking the lowest. These results indicate that although Copilot Chat generates the most correct and statically robust code, this comes at the cost of runtime efficiency.

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 can identify additional instances of performance regression that might have been overlooked if we had relied solely on execution time. Specifically, within the HumanEval dataset generated by Copilot, we find 37 and 3 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.

To further ensure that the identified performance regressions genuinely originate from algorithmic inefficiencies rather than superficial structural variations, we conducted a refined profiling analysis on the subset of generated code that exhibited significant performance regressions. Instead of profiling the entire script–which may include imports, initialization routines, or logging overhead–we restricted the analysis to the invocation of the verification function (`check(candidate)`), thereby capturing only the execution path of the generated implementation itself. For each profiled case, function-level statistics such as the number of calls, total self-time, and cumulative execution time were collected and ranked by cumulative cost. We further computed each function's relative contribution to the total runtime, quantifying the proportion of overall execution time attributable to each code component. By filtering out external library calls and module-level overhead, this analysis highlights the internal functions within the generated code that dominate runtime performance. The resulting critical-path profiles provide stronger causal evidence that the observed performance regressions are driven by inefficiencies inherent to the generated algorithms rather than incidental structural factors. This refinement complements the aggregate performance metrics reported earlier and enhances the validity of the prevalence analysis presented.

As shown in Table 4, the majority of execution time is concentrated in the task-specific function `filter_by_substring`, which accounts for approximately 75% of the total runtime within the critical path. Built-in operations such as `append` contribute only marginally. This observation indicates that the primary performance bottleneck stems from the generated function's internal logic rather than from external or structural code components, reinforcing the causal interpretation of the observed performance regressions. Furthermore, we have applied the same critical-path profiling procedure to all cases identified as perfor-

Table 4 Representative critical-path profiling result for a performance-regressed case (HumanEval_7 generated by Copilot)

Function	Location	Calls	Cumulative Time (s)	Proportion (%)
`check`	candidate file	1	6.79×10^{-5}	100.0
`filter_by_substring`	candidate file	120	5.08×10^{-5}	74.9
`append (list)`	built-in	270	1.50×10^{-5}	22.1
`disable (Profiler)`	built-in	1	5.84×10^{-6}	8.6

mance regressions across Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder on the HumanEval, MBPP, and EvalPerf datasets. The complete profiling results are included in our replication package.

The performance regressions identified in code generated by AI-generated code 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.

4.2 RQ2: What are the Root Causes in the Generated Code that Lead to Performance Regression?

Motivation In RQ1, we find that there are prevalent performance regressions in the AI-generated code. 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 not only provide valuable insights that can inform the development of coding assistants like Copilot but also potentially guide more AI-powered code generation models toward generating more efficient code. We can also inform developers with guidance on how to recognize potential performance pitfalls during the code generation process with coding assistants and code generation models.

Approach To investigate the underlying root causes responsible for performance regressions in AI-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. Given the observed performance regression instances across all four models, we conducted open coding for the generated code from all models. This approach enables a comprehensive identification of the root causes of performance regressions. Such an in-depth analysis not only provides targeted optimization insights for practical development scenarios but also serves as a valuable reference for improving the performance of code generation models in general. 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 (Zeng et al. 2019; Ding et al. 2020):

- **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. 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 0.87 (McHugh 2012).

Results We identify four major root causes of performance regressions from the code generated by all models. Our in-depth analysis of code samples exhibiting performance regressions yields four key categories of root causes at the code level, along with eleven sub-categories. These categories are detailed in Tables 5, 6, 7 and 8. Below, we discuss each root cause category with corresponding code examples for illustration.

4.2.1 R1 Inefficient Function Calls

Performance regression is often attributable to suboptimal function call choices, including the use of inefficient APIs, excessive recursion leading to deep stack issues, and unnecessary function abstraction.

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 Fig. 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 Fig. 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.

R1-3 Unnecessary Function Abstraction Simple operations are unnecessarily encapsulated into helper functions, leading to additional function call overhead without providing real modularity or reuse benefits. For example, in the code snippet shown in Fig. 6(c) from HumanEval-137 by Copilot Chat, the generated code defines a helper function *to_float()* only to replace commas with dots and cast to float. This abstraction is repeatedly invoked, while the same logic could be implemented in-line with significantly lower overhead. Such unnecessary function abstractions degrade runtime performance without offering meaningful software engineering advantages such as readability or reuse.

Table 5 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
EvalPerf	18	1	19	3	15	6	24	7	8	15	13	0	0	13

Table 6 Root causes of performance regression from the CodeLlama-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	1	0	1	0	0	0	0	4	0	4	2	0	0	2
AixBench	6	0	6	2	3	0	5	1	3	4	6	1	2	9
MBPP	49	4	53	1	8	1	10	14	11	25	3	0	0	3
EvalPerf	16	1	17	3	15	7	25	5	6	11	8	0	0	8

Table 7 Root causes of performance regression from the DeepSeek-Coder-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	2	3	5	1	0	2	3	1	2	3	1	1	0	2
AixBench	6	2	8	0	2	0	2	0	0	0	5	0	0	5
MBPP	37	4	41	4	16	1	21	19	6	25	3	0	0	3
EvalPerf	18	0	18	3	6	2	11	10	7	17	1	0	1	2

Table 8 Root causes of performance regression from the Copilot-Chat-generated Code

Dataset	Inefficient Function Calls				Inefficient Looping				Inefficient Algorithm			Inefficient Use of Language Features			
	Inefficient API usage	Excessive Recursion	Unnecessary Function Abstraction	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	14	0	3	17	0	1	15	16	2	9	11	9	0	0	9
AixBench	23	5	0	28	1	5	6	12	2	4	6	3	0	11	14
MBPP	109	4	0	113	1	15	15	31	53	4	57	2	0	0	2
EvalPerf	14	1	0	15	0	1	1	2	10	8	18	0	0	0	0

```java
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

```python
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

```python
def compare_one(a, b):
    def to_float(x):
        if isinstance(x, str):
            return float(x.replace(',', '.'))
        return float(x)
    fa, fb = (to_float(a), to_float(b))
    if fa == fb:
        return None
    return a if fa > fb else b
```

(c) Unnecessary Function Abstraction from HumanEval-137

Fig. 6 Examples of inefficient function calls

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

R2-2 Nested Looping The generated code exhibits issues with nested loops. In the code snippet shown in Fig. 7(b) from MBPP-12, a double loop is used for comparing and swap-

```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

ping 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.

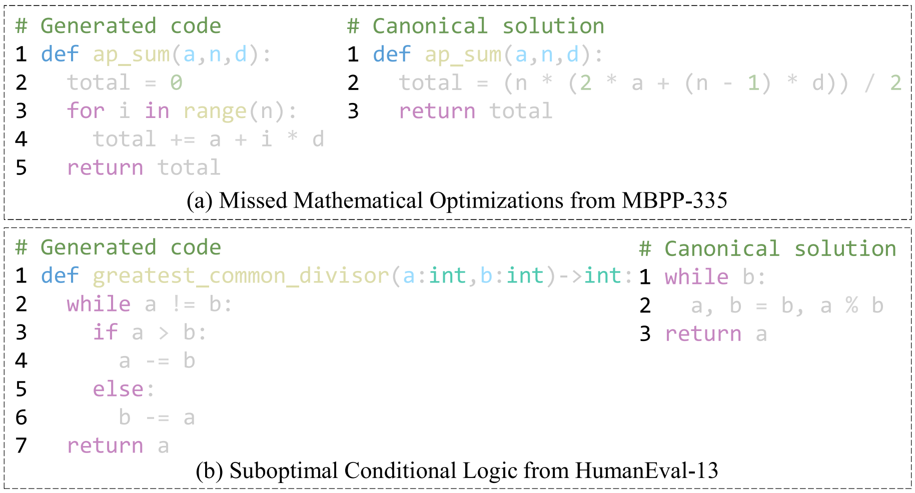
R2-3 Object Creation in Loops Creating objects repeatedly within loops can negatively impact performance due to extensive memory allocation and frequent garbage collection. In Fig. 7(c) from MBPP-363, the code example demonstrates a performance regression, where a new temp list is created within each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

4.2.3 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 sub-categories of inefficient algorithms.

R3-1 Missed Mathematical Optimizations The generated code may not always leverage mathematical optimizations to their full potential. In Fig. 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 Fig. 8(b) from Huma-



```
# Generated code
def ap_sum(a,n,d):
  total = 0
  for i in range(n):
    total += a + i * d
  return total
```

```
# Canonical solution
def ap_sum(a,n,d):
  total = (n * (2 * a + (n - 1) * d)) / 2
  return total
```

(a) Missed Mathematical Optimizations from MBPP-335

```
# Generated code
def greatest_common_divisor(a:int,b:int)->int:
  while a != b:
    if a > b:
      a -= b
    else:
      b -= a
  return a
```

```
# Canonical solution
while b:
  a, b = b, a % b
return a
```

(b) Suboptimal Conditional Logic from HumanEval-13

Fig. 8 Examples of inefficient algorithm

nEval-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.

4.2.4 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 Fig. 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 During the code generation process, unnecessary or redundant code is sometimes produced. For example, variables may be assigned values even though they are not subsequently read or utilized in the program. This not only increases the complexity of the code but can also affect its execution efficiency. In Fig. 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

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

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

(a) Underutilization of Language Features from MBPP-688

```python
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

```java
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

(such as in loops or core processing logic). In Fig. 9(c) from AixBench-72, the generated code contains issues with exception 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.

Root Causes of Performance Regressions Vary Across Models and Datasets Our analysis of the root causes of performance regressions (Table 5–8) reveals both model-specific characteristics and cross-model regularities in code generation. Across all evaluated models, inefficient function calls emerge as the most frequent source of performance regressions, particularly on the MBPP dataset, where Copilot (53), CodeLlama (49), DeepSeek-Coder (37), and Copilot Chat (109) all exhibit substantial API misuse. This suggests that LLMs tend to prioritize functional correctness and syntactic completeness over execution efficiency, reflecting limited reasoning about API complexity and call overhead. Inefficient algorithms, especially missed mathematical optimizations, form the second most frequent category, with substantial counts on MBPP–Copilot (76), CodeLlama (25), DeepSeek-Coder (25), and Copilot Chat (57). Inefficient looping appears less frequently but remains notable on MBPP, particularly in Copilot (66) and Copilot Chat (31). Together, these results indicate that current LLMs tend to prioritize functional correctness and structural completeness over runtime efficiency.

Performance-oriented datasets such as EvalPerf systematically expose optimization weaknesses across all models, despite their smaller size, with frequent issues in function calls (15–19), looping (2–25), and algorithms (11–18). In contrast, HumanEval shows minimal inefficiencies(typically single-digit instances per category), consistent with its simpler, correctness-focused tasks. The Java-based AixBench dataset uniquely reveals inefficiencies tied to static language features–particularly inefficient exception handling (0–11 instances per model) and unused variables (0–1)–which are largely absent in Python benchmarks. Python datasets, by contrast, primarily expose dynamic inefficiencies such as missed mathematical optimizations and nested looping, indicating insufficient exploitation of built-in optimizations like list comprehensions and generator expressions. These patterns reveal that LLMs face distinct challenges depending on the target language paradigm: static type systems in Java versus runtime performance idioms in Python.

At the model level, heterogeneous regression patterns highlight distinct performance characteristics across code generation. Although all models share the same high-level performance regression categories, their quantitative profiles differ markedly. The Copilot-based models (Copilot and Copilot Chat) exhibit the highest frequency of performance regressions, primarily driven by inefficient function calls and algorithmic inefficiencies. Specifically, on the MBPP dataset, Copilot Chat generates 113 code instances with function call issues and 57 instances with algorithmic issues, indicating that its enhanced completeness and modularity come at the cost of runtime efficiency. The model also introduces unique regressions like unnecessary function abstraction (appearing 3 times on HumanEval), indicating an overemphasis on structural modularity. Copilot follows a similar pattern with slightly fewer function call regressions (59) but higher looping inefficiencies (66 instances), suggesting that it tends to overuse iterative structures. In contrast, CodeLlama demonstrates a more balanced regression profile, with moderate counts across categories (53 instances with function call issues and 25 with algorithm issues on MBPP) but a specific weakness in nested looping on EvalPerf (15 instances), revealing limited awareness of loop-level computational overhead in performance-sensitive contexts. DeepSeek-Coder performs most robustly overall, exhibiting the lowest total regression counts (41 instances with function call issues and 25 algorithm issues on MBPP), indicating relatively efficient code generation. Nevertheless, even DeepSeek-Coder shows non-trivial regressions on performance-critical tasks (18 instances with function call issues and 17 with algorithm issues on EvalPerf), suggesting that no current model fully internalizes computational efficiency principles. Overall, these quantitative profiles reveal that while larger, instruction-tuned models like Copilot Chat achieve superior functional correctness, this improvement often coincides with higher performance regressions, reflecting a systematic trade-off between structural completeness and execution efficiency.

Quantitatively, inefficient API usages (ranging from 1–109 instances depending on model and dataset) and missed mathematical optimizations (0–53) dominate the distribution of regressions, accounting for over 70% of all identified cases, suggesting that models fundamentally struggle with reasoning about computational complexity and API overhead. In contrast, some inefficiency types follow a long-tail distribution–such as inefficient exception handling (appearing almost exclusively in Java code with 5–11 instances on AixBench but 0 on Python datasets), excessive recursion (0–6 instances), and unnecessary function abstraction (uniquely 3 instances in Copilot Chat on HumanEval)–that occur less frequently but can still introduce significant performance penalties when they occur, such as exponential time complexity or substantially degraded runtime performance. This distribution pattern provides clear priorities: enhanced training on API complexity reasoning and mathematical optimization patterns would address the most prevalent regressions, while targeted interventions for language-specific and context-specific long-tail patterns would handle high-impact edge cases.

While we employ static analysis tools (Qodana, SpotBugs, and PMD) and extend them with custom rules, we acknowledge that static rule matching is inherently imprecise. Many rules are based on code smells or heuristics, which do not always correspond to actual performance regressions. To assess the practical relevance of static warnings, we randomly sampled 50 flagged instances across all datasets and tools for manual validation. To ensure that our manual inspection of static warnings is representative, we employed a stratified and weighted sampling strategy. Specifically, we first stratified the warnings by static analysis tools (Qodana, SpotBugs, PMD) and datasets (HumanEval, MBPP, EvalPerf, AixBench). Within each stratum, we allocated samples proportionally to the number of warnings generated, while ensuring that each tool–dataset pair was represented by at least one instance. In total, 46 instances were selected following this proportional allocation. To further strengthen the validity of the evaluation, we additionally included 4 purposeful samples where static and dynamic analyses produced divergent results (e.g., static warnings not confirmed by profiling, or dynamic regressions missed by static tools). This yielded 50 instances in total. Each instance was independently inspected by two human experts, who judged whether the flagged code was likely to cause real performance degradation, considering both canonical code and available dynamic profiling results. Disagreements were resolved through discussion to reach a consensus. Our analysis revealed that only about 46% of the sampled warnings corresponded to real performance regressions, confirming the limited precision of static rule-based detection (Table 9).

Our manual inspection revealed that static analysis warnings differ in their practical relevance to performance, even within the same rule category. For example, the rule “Explicit return statement expected" sometimes flagged issues that led to inefficient execution paths (e.g., prolonged loop iterations), which were confirmed to cause measurable performance degradation. However, in other cases, the same warning merely indicated stylistic or correctness-related concerns without observable runtime impact. Similarly, warnings such as “Shadows built-in name ‘str’” or “Shadows name ‘x’ from outer scope" were purely cosmetic and unrelated to performance, while warnings like “Simplify chained comparison" or “Inconsistent return statements" were more consistently linked with inefficient code patterns. These findings highlight that static rules cannot be uniformly interpreted as indicators of performance regressions; their impact depends on context and requires careful validation.

Table 9 Distribution of 50 sampled static warnings for manual validation

Tool	Dataset	GitHub Copilot	CodeLlama	DeepSeek-Coder	Copilot Chat
Qodana	HumanEval	1	1	1	1
	MBPP	8	6	8	2
	EvalPerf	2	1	2	1
SpotBugs	AixBench	1	1	1	1
PMD	AixBench	2	2	1	3
Conflict cases (static vs. dynamic mismatch)		1	1	1	1
Total		15	12	14	9

Qodana is applied to HumanEval, MBPP, and EvalPerf; SpotBugs and PMD are applied to AixBench

Conflict cases are purposeful samples where static and dynamic analyses disagree

This underscores the importance of combining static analysis with dynamic profiling and manual inspection to distinguish truly performance-relevant issues from benign code smells (Table 10).

> 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.

4.3 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. Modifying the underlying architectures of code generation models such as GitHub Copilot,Copilot Chat, CodeLlama, and DeepSeek-Coder presents technical challenges. Instead, we investigated prompt engineering as a feasible approach to improving the performance characteristics of generated code. Our goal is to leverage prompts to guide code generation models in producing functionally correct and performance-optimized code. Prompt engineering does not require alterations to the internal mechanisms of the models, making it more practical for application to existing code generation systems. Furthermore, prompt engineering offers significant flexibility. By tailoring prompt content based on

Table 10 Manual validation results of 50 sampled static warnings

Tool	Validated samples	Confirmed (Yes)	False positives (No)	Precision
Qodana	38	16	22	42.1%
SpotBugs	4	2	2	50.0%
PMD	8	5	3	62.5%
Overall	50	23	27	46.0%

“Yes” indicates confirmed correlation with actual performance regressions, “No” indicates false positives insights from open coding analyses of root causes of performance regression, performance considerations can be more effectively integrated into the code generation process.

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

First, we designed few-shot prompts based on the four major root causes and eleven subcategories of performance regressions identified in RQ2. Specifically, these subcategories served as exemplar instances in our few-shot prompt, with each instance comprising an inefficient example (i.e., AI-generated code exhibiting performance regressions) and an efficient example (i.e., the corresponding human-written code). As shown in Fig. 10, the few-shot prompt consists of two primary components: a task description and a series of specific examples. The task description delineates the objective of performance optimization, while the accompanying specific examples meticulously highlight the observed performance regressions alongside their corresponding improvements, thus effectively guiding the code generation models to produce functionally correct and performance-optimized code.

Listing 1 CoT Prompt Example

```
# Write a function to implement the following
  requirements. Please provide only the code, without
  any other text.

# Chain of Thought: Performance-Optimized Implementation

I need to implement this function with optimal
  performance. Let me think step by step:

## Step 1: Understanding the Requirements
- What exactly does this function need to do based on
  the docstring and examples?
- What are the input constraints and expected outputs?
- What edge cases do I need to handle?

## Step 2: Algorithm Design Considerations
- What's the most efficient approach to solve this
  problem?
- Can I avoid unnecessary operations or memory
  allocations?
- Are there built-in functions or data structures that
  would be more efficient?
- What's the optimal time and space complexity I can
  achieve?

## Step 3: Performance Optimization Strategy
- How can I minimize the number of operations?
- Can I reduce memory usage or avoid creating temporary
  objects?
- Are there language-specific optimizations I can apply?
- Can I eliminate redundant computations?

## Step 4: Implementation
Now I'll read the following test code and the
  requirements and implement the most efficient
  solution and provide the code only, without any other
   text:
```

Prompt Template

Task Description:
Generate the given code snippets for great performance. Focus on:
1. Efficient function calls and API usage.
2. Minimizing inefficiencies in loops, such as reducing string concatenation.
3. Selecting optimal algorithms and avoiding redundant operations.
4. Leveraging built-in language features to improve runtime, memory usage, and CPU efficiency.

Example 1: Inefficient API Usage Optimization

```
# Inefficient Example
public void shuffle(int[] array) {
   for (int i = array.length - 1; i > 0; i--) {
      int rand = (int) (Math.random() * (i + 1));
          #Inefficient use of Math.random()
      int temp = array[i];
      array[i] = array[rand];
      array[rand] = temp;
   }
}

# Efficient Example
public void shuffle(int[] array) {
   Random random = new Random();
   for (int i = array.length - 1; i > 0; i--) {
      int rand = random.nextInt(i + 1); #Efficient use of
          Random.nextInt()
      int temp = array[i];
      array[i] = array[rand];
      array[rand] = temp;
   }
}
```

Example 2: Excessive Recursion Optimization

```
# Inefficient Example
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) # Excessive recursion

# Efficient Example
def is_simple_power(x, n):
   if (n == 1):
      return (x == 1)
   power = 1
   while (power < x):
      power = power * n
   return (power == x)
```

Fig. 10 Part of the few-shot prompt used in our prompt engineering for optimizing code performance

To further explore the effectiveness of prompt engineering in mitigating performance regressions, we additionally experimented with CoT prompts. CoT prompting has been widely used to enhance reasoning and decision-making in generative tasks (Wei et al. 2022). Our goal was to examine whether explicitly guiding models through a structured reasoning process could improve performance optimization in AI-generated code. The CoT prompt was designed to guide the model step by step through the process of understanding the requirements, considering algorithmic efficiency, and applying optimization strategies before producing the final implementation. As illustrated in Listing 1, the CoT prompt consists of four stages–requirement understanding, algorithm design considerations, performance optimization strategy, and implementation–while still enforcing code-only output for evaluation consistency. Compared with the exemplar-based few-shot prompt, the CoT prompt does not rely on specific inefficient–efficient code pairs but instead leverages structured reasoning instructions to encourage performance-aware generation.

Second, we applied these prompts to generate code using Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder across the HumanEval, AixBench, EvalPerf, and MBPP datasets. To ensure consistency, we maintained all model parameters from previous experiments, modifying only the prompt content. This approach maintained consistency in parameters such as temperature, *top_p*, and *max_gen_len*, ensuring fairness and reproducibility of the comparison results. Subsequently, we conduct both static and dynamic analyses of the generated code. Finally, we compare the performance metrics obtained in this stage with the previous baseline results to assess and compare the efficacy of the few-shot prompts and CoT prompts in improving the performance of the generated code. To maintain fairness in evaluation, we only analyzed scripts that successfully passed functional correctness checks under both few-shot and CoT prompting. As shown in Table 11, the number of passing scripts decreased due to potential compilation failures or test case failures post-prompting. This ensured that only scripts that compiled and passed all test cases in both prompting generations were included in the final analysis, providing an objective evaluation of the prompts' impact on performance optimization.

Results Table 11 presents the changes in performance regressions before and after applying CoT and few-shot prompt engineering to the Copilot, CodeLlama, DeepSeek-Coder, and Copilot Chat models. The results, which include both static and dynamic analyses, indicate that prompt engineering can lead to reductions in performance regressions, but the magnitude and direction of improvement depend on the specific model, dataset, and metric. For Copilot, the effects of prompt engineering varied across datasets and metrics. As shown in Table 11, both CoT and few-shot prompts influenced static and dynamic performance regressions, but no single prompt type consistently outperformed the other. For instance, in AixBench, which primarily involves static analysis, CoT eliminated regressions flagged by SpotBugs (1 $\rightarrow$ 0), whereas few-shot returned results similar to the baseline (1). For PMD, however, CoT increased regressions (15 $\rightarrow$ 25), while few-shot remained close to the baseline (15 $\rightarrow$ 14). In HumanEval, CoT substantially reduced static regressions (Qodana: 16 $\rightarrow$ 1), while few-shot also improved them, but to a lesser extent (16 $\rightarrow$ 10). Dynamic analysis revealed a different trend: few-shot produced clear benefits in execution time and CPU utilization (Execution time: 35 $\rightarrow$ 5; CPU: 45 $\rightarrow$ 28), and also reduced memory regressions (10 $\rightarrow$ 1), whereas CoT increased them (10 $\rightarrow$ 27). A similar divergence was observed in MBPP. CoT was most effective in reducing static regressions (Qodana: 228 $\rightarrow$ 36), but its dynamic

Table 11 Comparison of static and dynamic performance regression analysis before and after CoT prompt and few-shot prompt engineering for different models and datasets

				Static performance regression analysis			Dynamic performance regression analysis		
Model	Dataset	#Instances (intersection, %)	Prompt	SpotBugs	PMD	Qodana	Execution time	Memory usage	CPU Utilization
GitHub Copilot	HumanEval	114 (69.5%)	Before	N/A	N/A	16	35	10	45
			CoT	N/A	N/A	1	27	27	45
			Few-shot	N/A	N/A	10	5	1	28
	MBPP	699 (71.8%)	Before	N/A	N/A	228	127	57	268
			CoT	N/A	N/A	36	229	214	284
			Few-shot	N/A	N/A	176	55	56	225
	EvalPerf	96 (81.3%)	Before	N/A	N/A	23	67	30	48
			CoT	N/A	N/A	8	57	39	58
			Few-shot	N/A	N/A	16	69	31	46
	AixBench	62 (35.4%)	Before	1	15	N/A	N/A	N/A	N/A
			CoT	0	25	N/A	N/A	N/A	N/A
			Few-shot	1	14	N/A	N/A	N/A	N/A
CodeLlama	HumanEval	92 (56.1%)	Before	N/A	N/A	7	12	1	30
			CoT	N/A	N/A	7	6	2	11
			Few-shot	N/A	N/A	7	7	1	12
	MBPP	637 (65.4%)	Before	N/A	N/A	154	97	38	188
			CoT	N/A	N/A	167	99	24	186
			Few-shot	N/A	N/A	173	63	15	58
	EvalPerf	85 (72.0%)	Before	N/A	N/A	14	64	39	42
			CoT	N/A	N/A	16	76	37	40
			Few-shot	N/A	N/A	16	62	35	41
	AixBench	58 (33.1%)	Before	8	37	N/A	N/A	N/A	N/A
			CoT	3	29	N/A	N/A	N/A	N/A
			Few-shot	6	37	N/A	N/A	N/A	N/A
DeepSeek-Coder	HumanEval	42 (25.6%)	Before	N/A	N/A	1	17	6	13
			CoT	N/A	N/A	1	7	9	10
			Few-shot	N/A	N/A	1	4	4	6

Table 11 (continued)

Model	Dataset	#Instances (intersection, %)	Prompt	Static performance regression analysis			Dynamic performance regression analysis		
				SpotBugs	PMD	Qodana	Execution time	Memory usage	CPU Utilization
	MBPP	634 (65.1%)	Before	N/A	N/A	186	89	29	213
			CoT	N/A	N/A	145	48	35	80
			Few-shot	N/A	N/A	114	28	7	81
	EvalPerf	67 (56.8%)	Before	N/A	N/A	14	50	23	31
			CoT	N/A	N/A	10	60	23	34
			Few-shot	N/A	N/A	12	51	22	39
	AixBench	46 (26.3%)	Before	5	13	N/A	N/A	N/A	N/A
			CoT	1	12	N/A	N/A	N/A	N/A
			Few-shot	5	14	N/A	N/A	N/A	N/A
Copilot Chat	HumanEval	142 (86.6%)	Before	N/A	N/A	7	45	39	66
			CoT	N/A	N/A	20	55	56	75
			Few-shot	N/A	N/A	7	30	37	64
	MBPP	783 (80.4%)	Before	N/A	N/A	55	374	261	342
			CoT	N/A	N/A	48	398	274	371
			Few-shot	N/A	N/A	56	293	262	342
	EvalPerf	104 (88.1%)	Before	N/A	N/A	3	79	36	59
			CoT	N/A	N/A	4	74	38	62
			Few-shot	N/A	N/A	3	70	37	62
	AixBench	104 (59.4%)	Before	5	55	N/A	N/A	N/A	N/A
			CoT	5	52	N/A	N/A	N/A	N/A
			Few-shot	3	46	N/A	N/A	N/A	N/A

performance degraded considerably, with regressions increasing for memory (57 → 214) and CPU (268 → 284). By contrast, few-shot reduced regressions more consistently across dynamic metrics (Execution time: 127 → 55; Memory: 57 → 56; CPU: 268 → 225). Finally, in EvalPerf, the results were again mixed. CoT reduced static regressions more strongly (Qodana: 23 → 8) than few-shot (23 → 16), while dynamic metrics showed smaller differences: CoT slightly improved execution time (67 → 57), and few-shot marginally reduced CPU regressions (48 → 46), with memory remaining relatively stable across prompts. For Copilot, CoT is generally more effective in reducing static performance regressions, particularly in HumanEval, MBPP, and EvalPerf, while few-shot tends to yield better results in dynamic performance metrics, especially execution time and CPU utilization.

Similarly, as illustrated in Table 11, CodeLlama exhibited mixed effects of prompt engineering across datasets. In HumanEval, both CoT and few-shot reduced dynamic regressions most effectively, particularly for execution time and memory. In MBPP, few-shot provided the most notable benefits for dynamic metrics, reducing regressions in execution time, memory, and CPU utilization (Execution time: 97 → 63; Memory: 38 → 15; CPU: 188 → 58), while CoT had only minor effects. In EvalPerf, differences were less pronounced: few-shot slightly improved memory and CPU regressions (Memory: 39 → 35; CPU: 42 → 41), whereas CoT increased execution time regressions (64 → 76). In AixBench, static analysis showed that CoT reduced regressions flagged by both SpotBugs (8 → 3) and PMD (37 → 29), while few-shot achieved smaller reductions for SpotBugs (8 → 6) and no improvement for PMD (37). These results indicate that CoT is more effective for static performance regressions in CodeLlama, especially in AixBench, while few-shot tends to deliver stronger improvements in dynamic performance regressions, particularly in MBPP.

For DeepSeek-Coder, the effects of prompt engineering were also dataset- and metric-dependent, as summarized in Table 11. In HumanEval, both CoT and few-shot reduced dynamic regressions, with few-shot yielding the most consistent improvements (Execution time: 17 → 4; Memory: 6 → 4; CPU: 13 → 6). In MBPP, prompt engineering again proved effective: CoT and few-shot both reduced execution time regressions relative to the baseline (186 → 145 and 186 → 114, respectively), while memory and CPU regressions were substantially lower with few-shot (Memory: 89 → 28; CPU: 213 → 81). In EvalPerf, the trends were more nuanced. CoT reduced static regressions flagged by Qodana (14 → 10), but at the dynamic level it increased execution time regressions (50 → 60) and did not improve memory usage (23 → 23). Few-shot achieved only a slight reduction in memory regressions (23 → 22), while execution time (51) and CPU regressions (39) remained above baseline, indicating limited benefits for runtime efficiency. In AixBench, CoT outperformed few-shot in static analysis, decreasing SpotBugs regressions (5 → 1) and maintaining PMD at similar levels (13 → 12), whereas few-shot failed to reduce regressions (SpotBugs: 5; PMD: 14).

For Copilot Chat, the effects of prompt engineering were mixed across datasets (Table 11). In HumanEval, few-shot prompts improved dynamic performance, reducing execution time regressions from 45 to 30 and CPU regressions from 66 to 64, while CoT increased regressions across all dynamic metrics (execution time: 45 → 55; memory: 39 → 56; CPU: 66 → 75). In MBPP, few-shot yielded moderate gains by lowering execution time regressions (374 → 293), though memory and CPU regressions remained largely unchanged; by contrast, CoT increased total regressions, suggesting limited robustness of this strategy. In EvalPerf, both prompting strategies showed small improvements in dynamic performance. Few-shot slightly reduced execution time (79 $\rightarrow$ 70) and memory regressions (36 $\rightarrow$ 37 remained nearly stable), while CoT decreased execution time regressions to 74 but increased CPU regressions (59 $\rightarrow$ 62). For AixBench, prompt engineering was more effective in static analysis. Few-shot reduced SpotBugs regressions (5 $\rightarrow$ 3) and PMD regressions (55 $\rightarrow$ 46), whereas CoT achieved smaller gains (SpotBugs: 5; PMD: 55 $\rightarrow$ 52). Copilot Chat exhibited notable sensitivity to prompt engineering: while few-shot prompts offered tangible improvements in reducing dynamic regressions in HumanEval and MBPP, and static regressions in AixBench, CoT often introduced additional inefficiencies, underscoring the importance of selecting appropriate prompting strategies.

Few-shot prompt engineering consistently demonstrates the most robust improvements across dynamic performance metrics (execution time, memory, CPU) for Copilot, DeepSeek-Coder, and to a lesser extent CodeLlama and Copilot Chat. And CoT prompts yield mixed results: sometimes improving memory or CPU usage but occasionally increasing execution time regressions. HumanEval and MBPP benefit most from prompt strategies, while EvalPerf and AixBench show more dataset-dependent outcomes. Overall, prompt engineering–especially few-shot–proves broadly effective for reducing performance regressions across models and datasets, with Copilot and DeepSeek-Coder showing the most substantial gains.

Prompt Engineering is Most Effective 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.

Prompt engineering emerges as a valuable technique for mitigating performance regressions in AI-generated code. Few-shot prompt engineering specifically targeting the root causes of performance regressions facilitates the identification of potential performance bottlenecks and guides models to generate more efficient code. While execution time and CPU utilization benefit greatly from prompt engineering, memory usage improvements require further research to understand and address the underlying reasons.

5 Discussion

In this section, we discuss five aspects of AI-generated code performance: performance regressions across languages and datasets, prompt engineering for performance, energy and carbon footprint considerations, model-specific differences and architectural implications, and directions for future research on AI-generated code performance.

Performance Regressions of AI-Generated Code Across Languages and Datasets While AI-generated code can be functionally correct, it often exhibits performance regressions compared to human-written code, particularly when evaluated on performance-oriented benchmarks. Additionally, all models maintain relatively high correctness on HumanEval and MBPP, with pass rates typically above 70%. However, when assessed on the performance-oriented EvalPerf dataset, the magnitude of performance regressions increases across all models despite comparable functional accuracy, suggesting that current LLMs prioritize syntactic and semantic correctness over execution efficiency. In contrast, the AixBench dataset–comprising Java tasks–shows the lowest correctness rates (mostly between 37%–65%) and a higher incidence of static performance regressions detected by PMD and SpotBugs. Compared with the Python-based datasets (HumanEval, MBPP, and EvalPerf), where static issues are less prevalent, Java code generated by models such as CodeLlama and Copilot Chat contains more inefficient structures and performance-related anti-patterns. These findings indicate that Java tasks are more susceptible to static-level inefficiencies, likely due to the increased syntactic rigidity and verbosity of Java. The discrepancy 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. To gain a comprehensive understanding of performance regressions of AI-generated code across languages and datasets, we extended the scope of analysis from the initial focus on GitHub Copilot to include CodeLlama, DeepSeek-Coder and Copilot Chat, ensuring that our conclusions reflect the broader state of code generation technologies. We can more thoroughly evaluate the performance regression of different models and uncover their commonalities and differences in generating code across various programming languages and datasets. The datasets used in this study–HumanEval, MBPP, EvalPerf, and AixBench–provide complementary perspectives on code generation quality. While HumanEval and MBPP primarily assess functional correctness, EvalPerf focuses explicitly on performance-critical code. The datasets 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. Future work should expand performance-oriented datasets and incorporate more realistic, resource-intensive scenarios to better evaluate the computational robustness of AI-generated code.

Prompt Engineering for the Performance of AI-Generated Code Few-shot prompt engineering has proven effective in reducing performance regressions, but it does not entirely eliminate them. This suggests that existing AI models may require more complex architectures to better identify, predict, and prioritize performance optimization opportunities. In our study, we used few-shot prompts based on the root causes of performance regressions to guide the code generation models in producing more efficient code and conducted baseline evaluations. In addition to these exemplar-based prompts, we further incorporated CoT prompting, a widely studied strategy for eliciting step-by-step reasoning. The comparative results reveal that prompt type plays a crucial role in shaping model behavior: CoT tends to improve outcomes in static regression analyses (e.g., HumanEval, MBPP), while few-shot exemplar prompts more consistently enhance dynamic performance metrics such as execution time and CPU utilization. These findings indicate that prompt design is not a one-size-fits-all solution but instead interacts with dataset characteristics and model families. Future research should therefore explore systematic prompt design, including controlled ablation studies on exemplar count, selection strategy, and formatting style, as well as comparative analyses of advanced prompting paradigms such as instruction-tuned or negative-example prompts (White et al. 2023). Such directions can provide a more principled framework for assessing prompt robustness and for further enhancing the performance of AI-generated code.

Energy and Carbon Footprint of AI-Generated Code Performance evaluation is increasingly extending beyond traditional runtime and memory metrics to encompass energy consumption and carbon footprint, which are critical for sustainable software engineering. Inefficient code can result in unnecessary power draw and higher greenhouse gas emissions, particularly when deployed at scale or executed repeatedly in production environments (Vartziotis 2024; Shi et al. 2025; Sikand et al. 2024). While our analysis focused on execution time, memory usage, and CPU utilization, it provides an indirect foundation for understanding energy implications. Execution time and CPU utilization are strongly correlated with energy consumption: the performance regressions we identified–such as inefficient API usage, nested looping, and missed mathematical optimizations–not only degrade runtime efficiency but also amplify energy consumption through prolonged or computationally intensive execution. In this context, our findings that prompt engineering, particularly few-shot prompting, helps reduce execution time and CPU utilization across multiple datasets and models suggest that such strategies may also yield energy savings. By guiding models toward more efficient algorithmic patterns, performance-aware prompting can potentially reduce both computational overhead and its associated energy footprint, highlighting an environmental benefit that extends beyond immediate user experience. Future work should incorporate explicit energy efficiency metrics using tools such as PowerAPI, PyJoules into performance evaluations. Additionally, exploring prompt strategies explicitly designed for energy-efficient code generation and developing benchmark datasets that stress-test energy efficiency would complement existing correctness- and performance-oriented evaluations.

Model-Specific Differences and Architectural Implications Our comparative evaluation across GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder reveals that model architecture, training objectives, and decoding strategies jointly shape the observed performance characteristics. DeepSeek-Coder demonstrates comparatively strong static code quality, likely owing to its extensive pre-training on large-scale and diverse programming corpora emphasizing syntactic correctness and conventional implementation patterns. However, its dynamic performance occasionally lags behind, suggesting that the model's optimization objectives prioritize functional completeness and correctness over runtime efficiency. Moreover, its relatively constrained token budget may limit deeper reasoning within a single generation cycle, resulting in implementations that are correct but less optimized in execution. CodeLlama exhibits a more balanced performance profile across benchmarks, benefitting from its larger parameter scale and extended context window, which enhance its ability to process complex prompts and incorporate few-shot exemplars. Nevertheless, this flexibility can lead to output variability, particularly on heterogeneous datasets such as MBPP. Its fine-tuning focus on code comprehension rather than aggressive optimization may also account for its moderate static violations and stable but not outstanding runtime performance. A notable divergence is observed between the effects of few-shot and CoT prompting strategies across model families. Few-shot prompting–providing concrete exemplars of efficient implementations–consistently enhances performance for code-oriented models such as GitHub Copilot, CodeLlama, and DeepSeek-Coder, which are trained primarily under completion-style objectives and thus effectively leverage pattern-based generalization. In contrast, CoT prompting, which encourages explicit reasoning sequences, tends to yield limited or even adverse effects on these models, suggesting that low-level code synthesis tasks benefit less from abstract reasoning chains. Models with extended context windows, such as CodeLlama, demonstrate greater capacity to exploit few-shot exemplars by retaining and abstracting multiple efficiency-oriented code structures simultaneously. Conversely, models with stronger reasoning-oriented architectures, such as Copilot Chat, exhibit higher robustness to CoT prompting, although their improvements remain moderate in magnitude. This differential sensitivity underscores that prompt engineering effectiveness is inherently model-dependent, reflecting the degree to which each architecture aligns with reasoning-oriented versus pattern-imitation paradigms. The service-based models–GitHub Copilot and Copilot Chat–illustrate another dimension of architectural influence. Their proprietary nature and continuous deployment enable the integration of advanced post-processing mechanisms, broader and more recent training corpora, and reinforcement strategies aimed at improving alignment with user intent. Copilot Chat, powered by GPT-4.1, achieves the highest functional correctness and static code quality, indicating superior internal reasoning and error-detection capabilities. However, its elevated dynamic regression rate suggests a tendency to favor generalizable and feature-complete solutions over highly optimized implementations–an expected trade-off stemming from instruction-tuning and conversational fine-tuning objectives. Overall, these model-specific observations indicate that performance regressions in AI-generated code stem not solely from dataset characteristics but from the interplay among architectural scale, fine-tuning orientation, decoding strategy, and responsiveness to prompt design. A deeper understanding of these internal mechanisms provides valuable insight for future model development, emphasizing the importance of incorporating performance-awareness into pre-training objectives and decoding strategies to achieve a more balanced trade-off between functional correctness and runtime efficiency.

Future Research on AI-Generated Code Performance We investigate the performance regressions of code generated by GitHub Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder, which are representative of state-of-the-art code generation LLMs. The inclusion of multiple models broadens the scope of our findings, suggesting that the observed performance regressions may be a common challenge for LLMs utilizing similar architectures and methodologies. By employing both static and dynamic analysis techniques, we have provided a comprehensive evaluation of performance regressions and their underlying causes. 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. First, expanding performance evaluation to encompass a broader spectrum of programming languages, paradigms, and execution environments would clarify whether observed regressions arise from language-specific characteristics or reflect more fundamental architectural constraints. Such cross-environmental analyses could further disentangle the interplay between compiler optimizations, runtime systems, and structural properties of generated code. Second, continued exploration of prompt engineering strategies is essential for mitigating performance inefficiencies. Beyond few-shot and CoT prompting, systematic investigations into exemplar selection, formatting consistency, and adaptive or feedback-driven prompt design–potentially guided by empirical profiling of performance bottlenecks–may yield more robust frameworks for steering models toward efficiency-aware code generation. Third, tighter integration of performance feedback into model training and inference processes holds promise for bridging the persistent divide between functional correctness and runtime efficiency. Incorporating execution-aware training objectives, reinforcement signals derived from profiling data, or iterative runtime-guided refinement could enable models to internalize and optimize for performance trade-offs during generation. Finally, the development of standardized, performance-oriented benchmarks that jointly capture static and dynamic dimensions of efficiency would enable reproducible and fine-grained evaluation. Such benchmarks could support causal analyses of performance regressions, allowing researchers to isolate algorithmic inefficiencies from systemic overhead with greater precision. Collectively, these research directions extend the insights of our current study–spanning cross-language variability, prompt responsiveness, and architectural diversity–and chart a path toward next-generation, performance-conscious code generation systems capable of balancing correctness, readability, and computational efficiency.

6 Threats

In this section, we discuss the threats to the validity of our research.

External Validity The generalizability of our findings may be affected by the choice of datasets and code generation models. Our findings are based on four open-source datasets, i.e., HumanEval, AixBench, MBPP, and EvalPerf. 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 such as C++ or Rust. Performance regressions may significantly differ due to language features, compiler optimizations, and running environments. While GitHub Copilot, CodeLlama, DeepSeek-Coder, and Copilot Chat are representative code generation models, our findings may not apply universally to all LLMs. These LLMs differ in training data composition, architectural design, and decoding strategies, which may influence the observed patterns of performance regressions. Additionally, our study adopts one widely used configuration per model (e.g., CodeLlama-7B, DeepSeek-Coder-6.7B) and fixed decoding parameters (temperature = 0.0, top-p = 0.9) to ensure fairness and comparability across experiments. However, it is worth acknowledging that variations in model scale, sampling temperature, or decoding strategy can substantially alter generation diversity, determinism, and, consequently, performance efficiency. For instance, larger models may capture more nuanced optimization patterns but also incur greater resource overhead, whereas higher-temperature sampling might promote algorithmic creativity at the cost of runtime stability. Similarly, proprietary systems such as Copilot Chat expose configurable parameters (e.g., model selections, completion length, contextual window) that may influence generated code quality and efficiency. Future studies could systematically explore model variants, decoding strategies, and hyperparameter settings to quantify their effects on performance regressions to construct a more comprehensive understanding of how architectural and configurational factors influence the performance of AI-generated code.

Internal Validity To detect potential performance regressions, we employ three static analysis tools (e.g., Qodana, Spotbugs) and three dynamic profilers (e.g., cProfile). These tools were chosen for their established use in prior research and their robust support for the programming languages. While these tools provide robust coverage, several sources of bias may remain. Static analyzers rely on rule-based heuristics that can generate false positives unrelated to actual runtime inefficiencies. Conversely, dynamic profilers depend on the representativeness of the test inputs and can introduce profiling overhead. To mitigate these threats, all dynamic measurements were repeated ten times in a clean execution environment, with the first warm-up run discarded. We applied the Mann–Whitney U test and Cliff's δ to evaluate the significance and magnitude of performance differences, thereby improving the reliability of observed effects. To further refine our dynamic analysis, we introduced an additional refinement stage focusing on the critical execution path of each regression case. Instead of profiling entire scripts–which may include initialization, imports, or logging overhead–we restricted profiling to the execution of the verification function to isolate the core algorithmic behavior of generated code. This refinement mitigates confounding factors arising from superficial structural variations and enables a more causal interpretation of performance differences. Although this approach substantially improves interpretability, it remains limited to single-function tasks and may not fully capture multi-module interactions. 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 0.87, indicating considerable agreement (McHugh 2012). 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 incorporate more sophisticated instrumentation techniques, standardized function bodies for cross-language comparison, and finer-grained profiling mechanisms that capture multi-function dependencies. Expanding these methodological refinements would help disentangle algorithmic inefficiencies from structural or environmental noise, yielding a clearer causal understanding of performance regressions in AI-generated code.

Construct Validity Dynamic analysis using profilers can be influenced by environmental factors and noise. To mitigate this issue, we executed the generated code multiple times in a clean environment and applied the Mann-Whitney U test and Cliff's δ to analyze the ten repeated runs, aiming to better capture the actual performance of the code. However, some noise is inherent in performance monitoring. Future studies could consider increasing repetitions based on time and resource constraints. In addition, incorporating refined, function-level profiling could further reduce measurement bias by isolating algorithmic inefficiencies from structural overhead. Moreover, static analysis–though scalable and widely adopted–relies on rule-based heuristics that may not fully capture real performance regressions. Tools such as Qodana, SpotBugs, and PMD identify code smells that do not always result in measurable slowdowns at runtime. To assess the validity of these rule-based detections, we conducted a manual inspection of 50 randomly sampled static warnings across datasets and tools. The analysis revealed that approximately 46% of the flagged cases corresponded to genuine performance regressions when cross-checked via dynamic profiling. This finding highlights that static analysis serves as an effective large-scale screening mechanism but with limited precision due to the generality of rule definitions. Accordingly, our study integrates both static and dynamic evidence to ensure a more reliable and comprehensive assessment of performance regressions in AI-generated code. Future research can explore additional techniques or tools to uncover more complex performance problems within AI-generated code.

Conclusion Validity In our study, potential threats arise from performance variability across datasets and model architectures, as well as the stochastic nature of LLMs. Moreover, differences in runtime environments and the limited number of samples per dataset may influence the stability of the conclusions. To mitigate these issues, we report statistical significance, effect sizes, and variance indicators (standard deviations and confidence intervals) to quantify both the strength and consistency of observed differences. We also employ repeated executions for each generated code sample to reduce random noise and ensure stable measurements. Furthermore, by evaluating both closed-source (e.g., GitHub Copilot) and open-source models with transparent configurations, and conducting controlled experiments comparing few-shot and CoT prompting strategies, we strengthen the interpretability and reproducibility of our findings. Nonetheless, residual uncertainty remains due to inherent randomness in LLM outputs and the limited coverage of benchmark tasks. Future research could enhance conclusion validity through larger-scale replication studies and longitudinal analyses across LLM versions to assess temporal consistency and causal robustness.

7 Related Work

In this section, we discuss research related to this work, including AI-assisted code generation, assessing the quality of code generation techniques, and code performance analysis.

7.1 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 (Ling et al. 2016; Yin and Neubig 2017; Rabinovich et al. 2017; Iyer et al. 2018; Wei et al. 2019; Sun et al. 2020) and pre-trained model approaches (Feng et al. 2020; Guo et al. 2021; Ahmad et al. 2021; Wang et al. 2021; Nijkamp et al. 2023; Li et al. 2022). 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 (Drain et al. 2021; Hayati et al. 2018; Guo et al. 2019; Parvez et al. 2021; Zhou et al. 2023). This approach effectively reduces the decoding space, leading to improved quality in the generated code. Applying the NLP technologies like fine-tuning (Luo et al. 2024b; Wei et al. 2024) and reinforcement learning (Li et al. 2024a; Chae et al. 2024) to code generation models enhances the performance of generated code. Besides, the integration of AI-assisted code generation models into development environments has become a prominent trend in software development. Beyond well-known tools like GitHub Copilot, emerging solutions such as Cursor (2025) and Claude Code (2025) are actively advancing this field. These AI programming tools not only offer code generation and editing capabilities but also actively integrate with version control systems (e.g., Git) and continuous integration (CI) tools to optimize development workflows (Claude Code 2025).

7.2 Assessing the Quality of Code Generation Techniques

Many studies have evaluated the quality of code generation techniques or tools, e.g., ChatGPT, Copilot, CodeLlama, DeepSeek-Coder and CodeWhisperer, primarily focusing on whether these tools and models produce code that fulfills its intended function. Studies like Yetistiren et al. (2022) highlighted GitHub Copilot's ability to generate valid code with a high success rate. Sobania et al. (2022) found no significant difference in correctness between Copilot and other approaches. Similarly, Nguyen and Nadi (2022) and Yetistiren et al. (2023) evaluate code correctness, efficiency, and overall quality, with Yetistiren et al. (2023) 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. (2023) 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. Lertbanjongngam et al. (2022) compared human-written code with AlphaCode-generated code, emphasizing the need for developer review to identify performance bottlenecks. Coignion et al. (2024) 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. (2024b) 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 and Ji (2024) found performance limitations in GPT-4-generated code, which can be partially addressed through optimization. Garg et al. (2023) proposed leveraging LLMs with prompt engineering to optimize code performance, showing effective improvements in addressing performance regressions. Moreover, a growing number of performance-centric evaluation benchmarks, e.g., EffiBench (Huang et al. 2024b), Mercury (Du et al. 2024), and ECCO (Waghjale et al. 2024), have emerged. Concurrently, certain studies have explored the optimization of code performance using classification and reinforcement learning approaches (Seo et al. 2024; He et al. 2025). In addition, several recent works have proposed model-side improvements for efficiency. EffiLearner (Huang et al. 2024c) introduces a self-optimization framework that iteratively improves code efficiency through execution overhead profiling, demonstrating significant reductions in execution time across multiple models and benchmarks. Shypula et al. (2024) present Performance-Improving Edits, a method for learning performance-improving code edits from a curated dataset of over 77,000 human-made optimizations in competitive programming, enabling LLMs to suggest high-level algorithmic and API optimizations. ACECode (Yang et al. 2024) employs reinforcement learning with a dual-objective reward system to simultaneously optimize code efficiency and correctness in code LLMs, achieving notable improvements in both pass rates and runtime performance. PerfCodeGen (Peng et al. 2025) leverages execution feedback to guide LLMs toward more performant generations, using runtime profiling information to refine generated solutions iteratively. These works focus on optimizing model training and inference through various techniques, including self-optimization, reinforcement learning, execution feedback, and specialized fine-tuning strategies, whereas our study complements them by providing a systematic benchmarking and prompt-engineering perspective that evaluates performance regressions across multiple models and explores practical mitigation strategies accessible without model retraining. However, a comprehensive analysis of performance regressions in AI-generated code is still needed to identify root causes and guide further improvements.

7.3 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 (Chen et al. 2014; Reichelt et al. 2019a; van Dinten et al. 2024). For example, Gao et al. (2024) employed abstract syntax trees to represent the structural information of code, thereby identifying and comparing optimization patterns to support LLMs in more effectively optimizing code. Meanwhile, Venkatesh et al. (2024) investigated the application of LLMs in static analysis tasks, e.g., call graph analysis and type inference in Python programs. Many static analysis tools have also been proposed to analyze code performance regression. For instance, Qodana (JetBrains s.r.o 2024), developed by JetBrains, is a comprehensive static analysis engine that supports identifying a wide array of issues, including performance regressions. Other tools like Spotbugs (SpotBugs 2024) and PMD (PMD 2024) 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 (Reichelt et al. 2019b; Chen et al. 2023) and profiling (Yan et al. 2012; Weng et al. 2023). For example, Huang et al. (2024a) substantially improved the efficiency of llm-based code generation by integrating an adaptive optimization strategy informed by dynamic performance analysis. Beyond traditional runtime and memory concerns, performance is increasingly tied to energy consumption and environmental impact. Recent studies (Vartziotis 2024; Shi et al. 2025; Sikand et al. 2024) emphasize the importance of sustainable code generation, highlighting that inefficient code not only degrades user experience but also incurs unnecessary energy costs and carbon footprint. Islam et al. (2025) conducted a comprehensive study evaluating the energy efficiency of code generated by 20 popular LLMs and revealed that LLM-generated code often exhibits significantly higher energy consumption compared to human-written canonical solutions. Cursaru et al. (2024) performed a controlled experiment specifically examining Code Llama's energy efficiency, demonstrating systematic patterns of energy waste in AI-generated code. Furthermore, recent research has also begun to explore methods for optimizing the energy efficiency of LLM-generated code. Cappendijk et al. (2025) explored the effectiveness of different prompting strategies for generating energy-efficient code, finding that explicit energy-aware prompts can reduce energy consumption but with trade-offs in code complexity. Tools such as PowerAPI, psutil, and PyJoules provide promising means to integrate energy-related metrics into future evaluations of LLM-generated code. Nevertheless, a multifaceted approach encompassing static , dynamic, and sustainability-oriented evaluations remains essential for gaining a holistic understanding of AI-generated code quality, especially in the face of increasingly complex generative models and stringent performance requirements.

8 Conclusion

In this work, we analyzed the performance regression of code generated by LLM-based models, including GitHub Copilot, Copilot chat, CodeLlama, and DeepSeek-Coder. Our findings indicate that while these models are effective at generating functionally correct code, their output code exhibits significant performance regressions compared to human-written solutions. Our analysis revealed that the root causes of performance regressions are primarily at the code level, involving common root causes such as inefficient function calls and inefficient looping. To mitigate these performance regressions, we designed and implemented a few-shot prompting approach. This method leverages the identified root causes of code-level performance regressions to construct prompts with concrete examples, guiding the models to focus more on performance optimization during code generation. Our experimental results demonstrated that few-shot prompting can improve the performance of generated code, validating its potential as a mitigation strategy. In contrast, CoT prompting proved less effective–and in some cases detrimental–suggesting that reasoning-oriented strategies do not necessarily translate into improved performance regressions. Beyond individual datasets, we emphasized the importance of evaluating models under performance-critical conditions. Our analysis across both general-purpose and efficiency-oriented benchmarks highlights that performance regressions persist regardless of dataset scope, reinforcing the need for future evaluation frameworks that integrate efficiency as a first-class dimension of code quality. Overall, this study not only highlights the limitations of current LLMs in code generation performance but also underscores the need for further optimization of these models to meet the demands of performance-critical applications. While prompt engineering provides a viable means of mitigating certain performance regressions, its improvements remain inconsistent across tasks, suggesting that performance optimization requires integration at deeper levels of model design–particularly within training objectives, decoding strategies, and data curation pipelines. Future work should incorporate performance into model adaptation and pursue unified benchmarks that jointly assess functional, dynamic, and energy-related aspects of AI-generated code.

Acknowledgements This work is partially supported by the National Natural Science Foundation of China (Grant Nos. 62302347 and U2436205) and Open Project Funding of the Key Laboratory of Intelligent Sensing System and Security (Ministry of Education).

Author Contributions Shuang Li (First Author): Methodology, Data collection, Field study, Writing–draft preparation, review and editing. Yuntao Cheng (Second Author): Data curation, Writing–draft preparation. Jinfu Chen (Corresponding Author): Idea, Writing–review and editing, Writing–original draft preparation visualization, Investigation. Jifeng Xuan: Investigation and Methodology. Sen He: Investigation and Methodology. Weiyi Shang: Investigation and Methodology. All authors have read and agreed to the published version of the manuscript.

Funding This research received no external funding.

Data Availability Our data and script are open source and available at https://github.com/hebena/Performance-Analysis-of-AI-Generated-Code.

Declarations

Ethical Approval Not applicable.

Informed Consent Not applicable.

Conflict of Interest The authors declare no potential conflict of interest.

Clinical Trial Number Not applicable.

References

Ahmad WU, Chakraborty S, Ray B, Chang K (2021) 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, pages 2655–2668. Association for Computational Linguistics

Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q, et al (2021) Program synthesis with large language models. arXiv:2108.07732

Barke S, James MB, Polikarpova N (2023) Grounded copilot: How programmers interact with code-generating models. Proc ACM Program Lang 7(OOPSLA1):85–111

Cappendijk T, de Reus P, Oprescu A (2025) An exploration of prompting llms to generate energy-efficient code. In: 9th IEEE/ACM International Workshop on Green and Sustainable Software, GREENS@ICSE 2025, Ottawa, ON, Canada, April 29, 2025, pages 31–38. IEEE

Chae H, Kwon T, Moon S, Song Y, Kang D, Ong KT, Kwak B, Bae S, Hwang S, Yeo J (2024) Coffee-gym: An environment for evaluating and improving natural language feedback on erroneous code. In Y. Al-Onaizan, M. Bansal, and Y. Chen, editors, Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024, pages 22503–22524. Association for Computational Linguistics

Chen J, Shang W, Shihab E (2022) Perfjit: Test-level just-in-time prediction for performance regression introducing commits. IEEE Trans Software Eng 48(5):1529–1544

Chen J, Ding Z, Tang Y, Sayagh M, Li H, Adams B, Shang W (2023) 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, pages 845–857. ACM

Chen M, Tworek J, Jun H, Yuan Q, Pinto H.P dO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G et al (2021) Evaluating large language models trained on code. arXiv:2107.03374

Chen T, Shang W, Jiang ZM, Hassan AE, Nasser MN, Flora P (2014) 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, pages 1001–1012. ACM

Claude Code (2025) Claude code: The ai coding assistant. https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/overview. Accessed:2025-02-27

Coignion T, Quinton C, Rouvoy R (2024) A performance study of llm-generated code on leetcode. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, pages 79–89

Cursaru V, Duits L, Milligan J, Ural D, Sanchez BR, Stoico V, Malavolta I (2024) A controlled experiment on the energy efficiency of the source code generated by code llama. CoRR, abs/2405.03616

Cursor (2025) Cursor - The AI Code Editor. https://www.cursor.com/en. Accessed: 2025-02-27

Ding Z, Chen J, Shang W (2020) 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, pages 1435–1446. ACM

Drain D, Hu C, Wu C, Breslav M, Sundaresan N (2021) Generating code with the help of retrieved template functions and stack overflow answers. CoRR, abs/2104.05310

Du M, Luu AT, Ji B, Liu Q (2024) Ng S (2024) Mercury: A code efficiency benchmark for code large language models. In: Globersons A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak JM, Zhang C (eds) Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024. BC, Canada, December, Vancouver, pp 10–15

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) 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, volume EMNLP 2020 of Findings of ACL, pages 1536–1547. Association for Computational Linguistics

Fu Y, Liang P, Li Z, Shahin M, Yu J, Chen J (2025) Security weaknesses of copilot-generated code in github projects: An empirical study. ACM Transactions on Software Engineering and Methodology

Gao S, Gao C, Gu W, Lyu MR (2024) Search-based llms for code optimization. CoRR, abs/2408.12159

Garg S, Moghaddam RZ, Sundaresan N (2023) Rapgen: An approach for fixing code inefficiencies in zero-shot. arXiv:2306.17077

GitHub Docs (2025a) Asking GitHub copilot questions in your IDE. https://docs.github.com/en/copilot/how-tos/use-chat/use-chat-in-ide. Accessed: 2025-09-01

GitHub Docs (2025b) Github language support. https://docs.github.com/en/get-started/learning-about-github/github-language-support. Accessed: 2025-02-25

GitHub, Inc. (2025) Quickstart for GitHub Copilot. https://docs.github.com/en/copilot/get-started/quickstart?tool=vscode. Accessed: 2025-09-01

Guo D, Tang D, Duan N, Zhou M, Yin J (2019) 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, pages 855–866. Association for Computational Linguistics

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) 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

Guo D, Zhu Q, Yang D, Xie Z, Dong K, Zhang W, Chen G, Bi X, Wu Y, Li YK, Luo F, Xiong Y, Liang W (2024) Deepseek-coder: When the large language model meets programming - the rise of code intelligence. CoRR, abs/2401.14196

Hao Y, Li G, Liu Y, Miao X, Zong H, Jiang S, Liu Y, Wei H (2022) Aixbench: A code generation benchmark dataset. arXiv:2206.13179

Hayati SA, Olivier R, Avvaru P, Yin P, Tomasic A, Neubig G (2018) 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, pages 925–930. Association for Computational Linguistics

He P, Wang S, Chen T-H (2025) Codepromptzip: Code-specific prompt compression for retrieval-augmented generation in coding tasks with lms. arXiv:2502.14925

Hou W, Ji Z (2024) A systematic evaluation of large language models for generating programming code. arXiv:2403.00894

Huang D, Zeng G, Dai J, Luo M, Weng H, Qing Y, Cui H, Guo Z, Zhang JM (2024a) Effi-code: Unleashing code efficiency in language models. CoRR, abs/2410.10209

Huang D, Qing Y, Shang W, Cui H (2024) Zhang J (2024b) Effibench: Benchmarking the efficiency of automatically generated code. In: Globersons A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak JM, Zhang C (eds) Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024. BC, Canada, December, Vancouver, pp 10–15

Huang D, Dai J, Weng H, Wu P, Qing Y, Cui H, Guo Z (2024) Zhang J (2024c) Effilearner: Enhancing efficiency of generated code via self-optimization. In: Globersons A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak JM, Zhang C (eds) Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024. BC, Canada, December, Vancouver, pp 10–15

Huang Y, Li Y, Wu W, Zhang J, Lyu MR (2023) Do not give away my secrets: Uncovering the privacy issue of neural code completion tools. CoRR, abs/2309.07639

Islam MA, Jonnala DV, Rekhi R, Pokharel P, Cilamkoti S, Imran A, Kosar T, Turkkan BO (2025) Evaluating the energy-efficiency of the code generated by llms. CoRR, abs/2505.20324

Iyer S, Konstas I, Cheung A, Zettlemoyer L (2018) 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, pages 1643–1652. Association for Computational Linguistics

Jangali M, Tang Y, Alexandersson N, Leitner P, Yang J, Shang W (2023) Automated generation and evaluation of JMH microbenchmark suites from unit tests. IEEE Trans Software Eng 49(4):1704–1725

JetBrains s.r.o (2024) About Qodana. https://www.jetbrains.com/help/qodana/about-qodana.html. Accessed: 2024-12-30

Khoury R, Avila AR, Brunelle J, Camara BM (2023) 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, pages 2445–2451. IEEE

Laaber C, Leitner P (2018) 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, pages 119–130. ACM

Lertbanjongngam, S., Chinthanet, B., Ishio, T., Kula, R. G., Leelaprute, P., Manaskasemsak, B., Rungsawang, A., and Matsumoto, K. (2022). 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, pages 10–15. IEEE

Li B, Sun Z, Huang T, Zhang H, Wan Y, Li G, Jin Z, Lyu C (2024a) Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. Proc ACM Softw Eng 1(FSE):182–203

Li S, Cheng Y, Chen J, Xuan J, He S, Shang W (2024b) Assessing the performance of ai-generated code: A case study on github copilot. In: 35th IEEE International Symposium on Software Reliability Engineering, ISSRE 2024, Tsukuba, Japan, October 28-31, 2024, pages 216–227. IEEE

Li Y, Choi DH, Chung J, Kushman N, Schrittwieser J, Leblond R, Eccles T, Keeling J, Gimeno F, Lago AD, Hubert T, Choy P, de Masson d'Autume C, Babuschkin I, Chen X, Huang P, Welbl J, Gowal S, Cherepanov A, Molloy J, Mankowitz DJ, Robson ES, Kohli P, de Freitas N, Kavukcuoglu K, Vinyals O (2022) Competition-level code generation with alphacode. CoRR, abs/2203.07814

Ling W, Blunsom P, Grefenstette E, Hermann KM, Kociský T, Wang F, Senior AW (2016) 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

Liu J, Xie S, Wang J, Wei Y, Ding Y, Zhang L (2024a) Evaluating language models for efficient code generation. CoRR, abs/2408.06450

Liu,J. Xia CS, Wang Y, Zhang L (2024b) Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36

Luo W, Keung JW, Yang B, Ye H, Le Goues C, Bissyandé TF, Tian H, Le B (2024a) When fine-tuning llms meets data privacy: An empirical study of federated learning in llm-based program repair. CoRR, abs/2412.01072

Luo Z, Xu C, Zhao P, Sun Q, Geng X, Hu W, Tao C, Ma J, Lin Q, Jiang D (2024b) Wizardcoder: Empowering code large language models with evol-instruct. In: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net

Mayer L, Heumann C, Aßenmacher M (2024) Can opensource beat chatgpt? - A comparative study of large language models for text-to-code generation. CoRR, abs/2409.04164

McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia medica 22(3):276–282

Microsoft (2025) Visual Studio Code - Code Editing. https://code.visualstudio.com/. Accessed: 2025-09-01

Nguyen N, Nadi S (2022) 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, pages 1–5. ACM

Nijkamp E, Pang B, Hayashi H, Tu L, Wang H, Zhou Y, Savarese S, Xiong C (2023) 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

Niu L, Mirza MS, Maradni Z, Pöpper C (2023) 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, pages 2133–2150. USENIX Association

Parvez MR, Ahmad WU, Chakraborty S, Ray B, Chang K (2021) 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, pages 2719–2734. Association for Computational Linguistics

Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R (2022) 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, pages 754–768. IEEE

Peng Y, Gotmare AD, Lyu MR, Xiong C, Savarese S, Sahoo D (2025) Perfcodegen: Improving performance of LLM generated code with execution feedback. In: IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering, Forge@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025, pages 1–13. IEEE

PMD (2024) PMD: An extensible cross-language static code analyzer. https://pmd.github.io/. Accessed: 2024-04-25

Psutil Documentation Team (2025). Psutil documentation — psutil 7.1.0 documentation. https://psutil.readthedocs.io/en/latest/. Accessed: 2025-09-01

Python Software Foundation (2025a) psutil 7.0.0. https://pypi.org/project/psutil/. Accessed: 2025-09-01

Python Software Foundation (2025b) The Python Profilers. https://docs.python.org/3/library/profile.html. Accessed: 2025-09-01

Python Software Foundation (2025c) tracemalloc – Trace memory allocations. https://docs.python.org/3/library/tracemalloc.html. Accessed: 2025-09-01

Rabinovich M, Stern M, Klein D (2017) 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, pages 1139–1149. Association for Computational Linguistics

Reichelt DG, Kühne S, Hasselbring W (2019) On the validity of performance antipatterns at code level. Softwaretechnik-Trends 39(4):32–34

Reichelt DG, Kühne S, Hasselbring W (2019b) 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, pages 1146–1149. IEEE

Rozière B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu J, Remez T, Rapin J, Kozhevnikov A, Evtimov I, Bitton J, Bhatt M, Canton-Ferrer C, Grattafiori A, Xiong W, Défossez A, Copet J, Azhar F, Touvron H, Martin L, Usunier N, Scialom T, Synnaeve G (2023) Code llama: Open foundation models for code. CoRR, abs/2308.12950

Seo M, Baek J, Hwang SJ (2024) Rethinking code refinement: Learning to judge code efficiency. In: Al-Onaizan Y, Bansal M, Chen Y (eds) Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12–16, 2024. Association for Computational Linguistics, pp 11045–11056

Shi J, Yang Z, Lo D (2025) Efficient and green large language models for software engineering: Literature review, vision, and the road ahead. ACM Trans. Softw. Eng. Methodol. 34(5):1–22

Shypula A, Madaan A, Zeng Y, Alon U, Gardner JR, Yang Y, Hashemi M, Neubig G, Ranganathan P, Bastani O, Yazdanbakhsh A (2024) Learning performance-improving code edits. In: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net

Sikand S, Mehra R, Sharma VS, Kaulgud V, Podder S, Burden AP (2024) Do generative AI tools ensure green code? an investigative study. In: Proceedings of the 2nd International Workshop on Responsible AI Engineering, RAIE 2024, Lisbon, Portugal, 16 April 2024, pages 52–55. ACM

Sobania D, Briesch M, Rothlauf F (2022) 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, pages 1019–1027. ACM

SonarQube (2024) SonarQube: A code quality management platform. https://www.sonarqube.org/. Accessed: 2024-04-28

SpotBugs (2024) SpotBugs: Find bugs in Java Programs. https://spotbugs.github.io/. Accessed: 2024-04-25

Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L (2020) 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, pages 8984–8991. AAAI Press

Tanenbaum AS (2015) Modern Operating Systems. Pearson, Boston, fourth edition edition

van Dinten I, Derakhshanfar P, Panichella A, Zaidman A (2024) The slow and the furious? performance antipattern detection in cyber-physical systems. J Syst Softw 210:111904

Vartziotis T, Dellatolas I, Dasoulas G, Schmidt M, Schneider F, Hoffmann T, Kotsopoulos S, Keckeisen M (2024) Learn to code sustainably: An empirical study on green code generation. In: LLM4CODE@ICSE, pages 30–37

Venkatesh APS, Sabu S, Mir AM, Reis S, Bodden E (2024) The emergence of large language models in static analysis: A first look through micro-benchmarks. In: D. Lo, X. Xia, M. D. Penta, and X. Hu, editors, Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, FORGE 2024, Lisbon, Portugal, 14 April 2024, pages 35–39. ACM

Visual Studio Code Docs (2025) Github copilot in vs code settings reference. https://code.visualstudio.com/docs/copilot/reference/copilot-settings. Accessed: 2025-09-01

Waghjale S, Veerendranath V, Wang Z, Fried D (2024) ECCO: can we improve model-generated code efficiency without sacrificing functional correctness? In: Y. Al-Onaizan, M. Bansal, and Y. Chen, editors, Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024, pages 15362–15376. Association for Computational Linguistics

Wang Y, Wang W, Joty SR, Hoi SCH (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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, pages 8696–8708. Association for Computational Linguistics

Wang Z, Zhou Z, Song D, Huang Y, Chen S, Ma L, Zhang T (2025) Towards understanding the characteristics of code generation errors made by large language models

Wei B, Li G, Xia X, Fu Z, Jin Z (2019) 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, pages 6559–6569

Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D (2022) Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022

Wei Y, Cassano F, Liu J, Ding Y, Jain N, Mueller Z, de Vries H, von Werra L, Guha A (2024) Zhang L (2024) Selfcodealign: Self-alignment for code generation. In: Globersons A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak JM, Zhang C (eds) Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024. BC, Canada, December, Vancouver, pp 10–15

Weng L, Hu Y, Huang P, Nieh J, Yang J (2023) 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, pages 1–17. ACM

White J, Fu Q, Hays S, Sandborn M, Olea C, Gilbert H, Elnashar A, Spencer-Smith J, Schmidt DC (2023) A prompt pattern catalog to enhance prompt engineering with chatgpt. CoRR, abs/2302.11382

Yan D, Xu G, Rountev A (2012) Uncovering performance problems in java applications with reference propagation profiling. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pages 134–144. IEEE Computer Society

Yang C, Kang HJ, Shi J, Lo D (2024) Acecode: A reinforcement learning framework for aligning code efficiency and correctness in code language models. CoRR, abs/2412.17264

Yang Z, Zhao Z, Wang C, Shi J, Kim D, Han D, Lo D (2023) Gotcha! this model uses my code! evaluating membership leakage risks in code models. CoRR, abs/2310.01166

Yetistiren B, Ozsoy I, Tuzun E (2022) 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, pages 62–71. ACM

Yetistiren B, Özsoy I, Ayerdem M, Tüzün E (2023) Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. CoRR, abs/2304.10778

Yin P, Neubig G (2017) 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, pages 440–450. Association for Computational Linguistics

Zan D, Chen B, Zhang F, Lu D, Wu B, Guan B, Wang Y, Lou J-G (2022) Large language models meet nl2code: A survey. preprint arXiv:2212.09420

Zeng Y, Chen J, Shang W, Chen T-HP (2019) Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. Empir Softw Eng 24(6):3394–3434

Zhang B, Du T, Tong J, Zhang X, Chow K, Cheng S, Wang X, Yin J (2024) Seccoder: Towards generalizable and robust secure code generation. In: Y. Al-Onaizan, M. Bansal, and Y. Chen, editors, Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024, pages 14557–14571. Association for Computational Linguistics

Zheng Z, Ning K, Wang Y, Zhang J, Zheng D, Ye M, Chen J (2023) A survey of large language models for code: Evolution, benchmarking, and future trends. arXiv:2311.10372

Zhou S, Alon U, Xu FF, Jiang Z, Neubig G (2023) 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

Authors and Affiliations

Shuang Li[1] · Yuntao Cheng[1] · Jinfu Chen[1,2] · Jifeng Xuan[1] · Sen He[3] · Weiyi Shang[4]

✉ Jinfu Chen
jinfuchen@whu.edu.cn

Shuang Li
shuangli.cs@whu.edu.cn

Yuntao Cheng
cytzzz@whu.edu.cn

Jifeng Xuan
jxuan@whu.edu.cn

Sen He
senhe@arizona.edu

Weiyi Shang
wshang@uwaterloo.ca

[1] School of Computer Science, Wuhan University, Wuhan, China

[2] School of Computer Science & Key Laboratory of Intelligent Sensing System and Security (Ministry of Education), Wuhan University & Hubei University, Wuhan, China

[3] Department of Electrical and Computer Engineering, University of Arizona, Tucson, Arizona, USA

[4] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada