

Bachelor's Thesis

# Impact of Type Annotation and Usage Context on Copilot's Code Completion: An Empirical Study

Minh-Khue Pham

September 4, 2024

Advisor:

Dr. Norman Peitek    Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel

Chair of Software Engineering

Prof. Dr. Vera Demberg

Chair of Computer Science and Computational Linguistics

Chair of Software Engineering  
Saarland Informatics Campus  
Saarland University



UNIVERSITÄT  
DES  
SAARLANDES



## **Erklärung**

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

## **Statement**

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

## **Einverständniserklärung**

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

## **Declaration of Consent**

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 04.09.2024  
(Datum/Date)

  
(Unterschrift/Signature)



# Abstract

---

Large Language Models (LLMs) have demonstrated a tremendous impact across multiple domains, significantly transforming various aspects of academia and industry. In June 2022, GitHub and OpenAI launched Copilot, a code completion tool that exemplifies the practical application of LLMs in software engineering. Despite its extraordinary ability in code prediction, little is known about which factors affect Copilot’s suggested code, and no study has investigated the impact of type annotation and usage context on Copilot prediction. In this study, we propose to close this gap by evaluating the accuracy and complexity of suggested function bodies by Copilot, given four types of query contexts: no type annotation, no usage context; type annotation, no usage context; no type annotation, usage context; type annotation, usage context. Overall, we found that Copilot’s completions for CodeCheck problems have average accuracy and low complexity, with substantial variations between four query contexts. The amount of information in CodeCheck queries shows a positive correlation with the accuracy of Copilot’s predictions, while tending to negatively correlate with their complexity. LeetCode problems, on the other hand, have extremely high accuracy and low complexity, with no obvious variations between the four query contexts. These results allow us to better understand which input variables should be given to Copilot in order to improve the quality of its code predictions.



# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview . . . . .	2
<b>2</b>	<b>Background and Related Work</b>	<b>5</b>
2.1	Code Completion . . . . .	5
2.2	Large Language Models and GitHub Copilot . . . . .	7
2.3	Data Type and Type Annotations . . . . .	9
2.4	Usage Context . . . . .	10
<b>3</b>	<b>Methodology</b>	<b>11</b>
3.1	Research Questions . . . . .	11
3.2	Materials . . . . .	11
3.2.1	Programming Language: Python . . . . .	11
3.2.2	Programming Problems: CodeCheck and LeetCode . . . . .	12
3.3	Variables . . . . .	17
3.3.1	Independent Variables: Type Annotation and Usage Context . . . . .	17
3.3.2	Dependent Variables: Accuracy and Complexity . . . . .	18
3.4	Procedure . . . . .	19
<b>4</b>	<b>Results</b>	<b>23</b>
4.1	RQ1: Accuracy . . . . .	23
4.2	RQ2: Complexity . . . . .	26
<b>5</b>	<b>Discussion</b>	<b>31</b>
5.1	RQ1: Accuracy . . . . .	31
5.2	RQ2: Complexity . . . . .	32
<b>6</b>	<b>Threats to Validity</b>	<b>35</b>
6.1	Internal Validity . . . . .	35
6.2	External Validity . . . . .	36
<b>7</b>	<b>Concluding Remarks</b>	<b>37</b>
7.1	Conclusion . . . . .	37
7.2	Future Work . . . . .	37
<b>A</b>	<b>Appendix</b>	<b>39</b>
	<b>Bibliography</b>	<b>43</b>

# List of Figures

Figure 1.1	Copilot suggested an inappropriate incomplete function body . . . . .	2
Figure 2.1	Copilot's code completion . . . . .	5
Figure 2.2	Copilot's ranked list of possible completions . . . . .	6
Figure 2.3	Python code snippet without type annotations (left) and with type annotations (right) . . . . .	9
Figure 2.4	Usage context in Python . . . . .	10
Figure 3.1	TreeNode class . . . . .	12
Figure 3.2	ListNode class . . . . .	12
Figure 3.3	CodeCheck coding environment . . . . .	14
Figure 3.4	CodeCheck submission status . . . . .	14
Figure 3.5	LeetCode coding environment . . . . .	15
Figure 3.6	LeetCode submission statuses . . . . .	15
Figure 3.7	NONE: No type annotation, no usage context query . . . . .	18
Figure 3.8	TA: Type annotation, no usage context query . . . . .	18
Figure 3.9	CON: No type annotation, usage context query . . . . .	18
Figure 3.10	TA_CON: Type annotation, usage context query . . . . .	18
Figure 3.11	Example of Copilot's code completion . . . . .	19
Figure 3.12	Modify CodeCheck's coding environment to measure accuracy . . . . .	21
Figure 3.13	Modify LeetCode's coding environment to measure accuracy . . . . .	21
Figure 4.1	Similar Copilot's completions for NONE and TA, only return parameters are different . . . . .	25
Figure 4.2	Copilot's completion contains backtrack helper function . . . . .	26
Figure 4.3	SyntaxError: miss closing parenthesis . . . . .	26
Figure 4.4	SyntaxError: append type annotations on no-type-annotation query . . . . .	26
Figure 4.5	SyntaxError: redundant type annotations . . . . .	27
Figure 4.6	Cyclomatic complexity of Copilot's completions for 50 CodeCheck problems . . . . .	28
Figure 4.7	Cyclomatic complexity of Copilot's completions for 50 LeetCode problems . . . . .	28
Figure 4.8	Cognitive complexity of Copilot's completions for 50 CodeCheck problems . . . . .	29
Figure 4.9	Cognitive complexity of Copilot's completions for 50 LeetCode problems . . . . .	29
Figure 5.1	Similar Copilot's completions for TA and TA_CON, only compared strings are different . . . . .	32
Figure 5.2	Copilot's predicted arguments for usage context are identical to the example arguments on the LeetCode website . . . . .	32



# List of Tables

Table 4.1	Frequency of patterns in Copilot’s predictions . . . . .	24
Table A.1	Accuracy of Copilot’s completions for 50 CodeCheck problems . . .	40
Table A.2	Accuracy of Copilot’s completions for 50 LeetCode problems . . . .	41
Table A.3	Accuracy of Copilot’s completions for 37 LeetCode problems, for which Copilot predicted similar usage context arguments to the LeetCode website . . . . .	42

# Acronyms

- LLMs Large Language Models
- IDEs Integrated Development Environments
- NLMs Neural Language Models



# Introduction

---

Recent breakthroughs in deep learning have resulted in the growing prominence of LLMs that are capable of generating novel human-like content. GPT-3 [4], which has been developed utilizing the underlying deep learning technology used in LLMs, is able to comprehend and generate realistic text. DALL-E [31] is a text-to-image model that generates images based on textual prompts. This model can produce highly detailed images in multiple styles based on the user's instructions. In 2024, OpenAI released SORA [36], a text-to-video model that further developed DALL-E's capabilities from static images to dynamic videos. SORA has impressed the academic and industrial worlds with its realistic and inventive scenes generated from input text descriptions. Codex [7] is another AI model created by OpenAI that has been fine-tuned to comprehend and generate code, thereby boosting developer productivity.

Automatic code generation has long been a software engineer's fantasy. It reduces the need for remembering and manually coding, guarantees consistency and quality, and accelerates the development process. After less than one year in technical preview, in June 2022, GitHub and OpenAI released Copilot, an automatic code completion model claiming to be developers' "AI pair programmer". When provided with contexts, such as comments, method names, or surrounding code, Copilot can automatically suggest code completion in different programming languages. With its extraordinary performance and ability to interface with popular Integrated Development Environments (IDEs), such as Visual Studio Code, JetBrains, and Neovim, Copilot has created a "hype" among developers in the tech world and is the most likely AI assistant to be adopted by them.

As software projects grow in complexity and size, it becomes impractical for engineers to continue writing code from scratch. Knowing how to utilize an autonomous code generation model, such as Copilot, will be a significant advantage and possibly a required skill in the future. However, Copilot "can't give any assurance that the code is bug free" [15] and is heavily dependent on the input prompts. Figure 1.1 depicts an example of an inappropriate suggestion from Copilot. The expected solution is the correct function body for function `romanToNumber`, which converts input from a Roman character into its equivalent numeric value. However, Copilot generated an inappropriate incomplete function body in this example. Understanding the interplay between Copilot's input queries and output enables developers to modify the queries to acquire the desired code suggestions. Autonomous code generation models can drastically alter the way developers learn to program by requiring them to only understand how to generate appropriate queries rather than writing code from scratch. As these models can significantly improve productivity, reduce development costs, enhance code quality, and support software projects' scalability [15], the ability to design

```

1  def romanToNumber(input):|
    roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
    number = 0
    for i in range(len(input)):
        if i > 0 and roman[input[i]] > roman[input[i - 1

```

Figure 1.1: Copilot suggested an inappropriate incomplete function body

effective queries that generate correct predictions will have the potential to be a required human-computer interaction skill for developers in the future.

Given that the modern world is composed of billions of lines of code and the relevance of automatic code generation models in code production, a concrete investigation of variables affecting these models is merited. The goal of this thesis is to empirically assess the impact of type annotation and usage context on Copilot’s prediction. In particular, we evaluate Copilot by requesting it to predict function bodies for 100 Python programming problems given four types of query contexts: no type annotation, no usage context; type annotation, no usage context; no type annotation, usage context; type annotation, usage context. We then evaluate and compare the accuracy and complexity of four suggested variations within each problem. While preliminary, we aim to provide deeper insight with this study into how type annotation and usage context can change Copilot’s prediction, mitigate threats of validity, and lay the groundwork for future studies.

## 1.1 Overview

We evaluated Copilot’s ability to complete function bodies for programming problems from two public question pool websites, CodeCheck [10] and LeetCode [25]. These two websites provide us with the necessary information to generate comprehensive query contexts for Copilot, such as problem descriptions, function names, number of input and return parameters, and type annotations of input and return parameters. Since Copilot is non-deterministic and returns a ranked list of suggestions, we only took into account the first suggested function bodies. To evaluate the accuracy of Copilot’s suggested function bodies, we run against corresponding test cases that are already provided on the two websites. To evaluate the complexity of the suggested function bodies, we use SonarQube [41] to measure the cyclomatic and cognitive complexity. Overall, we tested Copilot’s prediction on 100 programming problems, including 50 from CodeCheck and 50 from LeetCode. For each problem, four queries were generated and evaluated, totaling 400 evaluations for 400 queries.

We found that Copilot’s predictions for LeetCode problems have high accuracy, low complexity, and small variation across four queries. In particular, the total number of test cases passed for all four queries exceeds 90%, with query contexts that include only type annotations generating suggestions that pass the highest number of test cases (95.5%). All four queries have the same median 4 of cyclomatic and cognitive complexity. On the contrary, Copilot predictions for CodeCheck problems are less accurate and have a greater variation over four queries. The amount of information is positively correlated with the

accuracy of Copilot's predictions and tends to be negatively correlated with their complexity. Function bodies generated by queries with no type annotation, no usage context passed the lowest percentage of total test cases (41.4%), while those generated by queries with both type annotation and usage context passed the most (72.8%). There is also a substantial variation in complexity between queries with no type annotation, no usage context and those with both. Specifically, suggestions for queries with no information have the highest cyclomatic and cognitive complexity (median 3 and 4, respectively), while queries containing both information have the lowest cyclomatic and cognitive complexity (median 2 and 1.5, respectively).

We also observed several notable patterns in Copilot's code completion, which will be described in more detail in [Chapter 4](#) (Results).



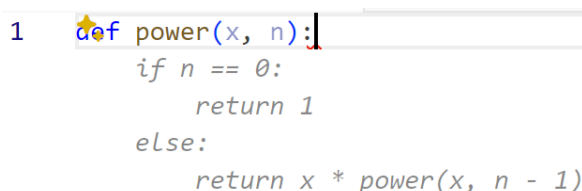
# Background and Related Work

In this chapter, we provide necessary background information and a brief summary of relevant research.

## 2.1 Code Completion

Code completion is an essential feature in IDEs that supports developers by suggesting completions for partially written code. It can predict and display in near-real-time possible completions based on the context and available code elements. Figure 2.1 shows an example of using the auto-code completion feature on GitHub Copilot. The suggested completion appears below the query after the cursor position and can be accepted using the Tab key. Copilot provides not only one completion, but also generates a ranked list of possible completions when pressing Ctrl + Enter, as shown in Figure 2.2. Code completion can optimize developer productivity by mitigating the need to implement code manually, preventing syntax errors, and speeding up the coding process. As the suggestions are displayed nearly in real-time, code completion relies heavily on compile-time type information to predict the next tokens based on the current context [43]. This works well for statically typed languages such as Java but is less supported for dynamically typed languages like Python, as type annotation is not mandatory in these languages. Several researchers have adopted various approaches to increase the accuracy of code completion in IDEs.

In a 2012 publication, Hindle et al. [19] proposed the "naturalness of software" hypothesis, arguing that real-world programs that are written by humans are actually simple, repetitive, and predictable despite the complexity of programming languages. This means they can leverage language models to capture predictable characteristics of programs and apply them to solve software engineering tasks. In particular, they employed the n-gram model to construct a simple code completion engine for Java, which successfully improved the IDEs' code completion ability.

A screenshot of a code editor showing a Python function definition. The first line is '1 def power(x, n):'. The cursor is at the end of this line. Below it, the function body is shown with indentation: 'if n == 0:', 'return 1', 'else:', and 'return x \* power(x, n - 1)'. The code is displayed in a light gray font on a white background.

```
1 def power(x, n):  
    if n == 0:  
        return 1  
    else:  
        return x * power(x, n - 1)
```

Figure 2.1: Copilot's code completion

## GitHub Copilot Suggestions

2 Suggestions

### Suggestion 1

```
def power(x, n):  
    if n == 0:  
        return 1  
    else:  
        return x * power(x, n-1)
```

Accept suggestion 1

### Suggestion 2

```
def power(x, n):  
    if n == 0:  
        return 1  
    return x * power(x, n-1)
```

Accept suggestion 2

Figure 2.2: Copilot's ranked list of possible completions

Tu et al. [43] conducted a follow-up study on the "localness of software", claiming that human-written programs have useful local regularities that can be captured. For example, they aimed to predict the next token in a sequence for `(int`. Assume that in the training data, 30% of the time `i` is the next token, while 5% of the time `size` is the next token, thus `i` is selected. This would be a reasonable choice in an isolated line. However, if multiple preceding lines in the same file contained `for (int size`, while none contained `for (int i`, the token `size` should be selected instead. To address this challenge, they exploited the localness of software by introducing a cache language model that included both `n`-gram and an additional cache. This new model performs significantly better in code completion tasks than the language model that only consists of `n`-gram.

However, the above-mentioned articles predict the next tokens based on a limited set of shallow features, namely the `n` code tokens that precede the prediction. Such features are a poor choice since they blindly capture dependencies that are only syntactically local to the token to be predicted by the model. Understanding this problem, Raychev et al. [33] proposed a different approach for the code completion task by using decision tree learning. The main idea is to recursively split the training data similarly to decision trees and then learn smaller, specialized probabilistic models for each branch of the tree. This approach served as the foundation for the development of the statistical code completion system, DEEP3. DEEP3 can predict the next tokens of dynamically typed languages such as JavaScript and Python with precision exceeding 82% and 69%, respectively, and significantly outperforms other previous approaches in overall prediction accuracy.

According to Karampatsis et al. [22], source code differs from natural language in that developers can freely generate new identifier names, leading to a larger and more



sophisticated vocabulary. All models trained on a large-scale software corpus must deal with an incredibly broad and sparse vocabulary and therefore have difficulty scaling beyond as few as a hundred projects. Another problem is that models cannot anticipate identifiers that do not appear in the training set. Given these two problems, they proposed an open-vocabulary Neural Language Models (NLMs) approach for code completion tasks that can both handle identifiers not seen in training and reduce vocabulary by three orders of magnitude, scaling to a hundred times larger corpora than previous NLMs approaches. The results demonstrated that the open-vocabulary NLMs outperform both n-gram language models and closed-vocabulary NLMs in the code completion task.

## 2.2 Large Language Models and GitHub Copilot

LLMs are self-supervised and semi-supervised learning artificial neural networks that have been trained on massive amounts of data to predict billions of parameters, hence the name "large". LLMs use deep learning in order to understand how characters, words, and sentences interact with one another, allowing them to recognize and interpret human language or other complicated data. LLMs are often further tailored to the particular tasks, such as interpreting questions and generating responses (e.g., GPT-3 [4]), generating images (e.g., DALL-E [31]) or videos (e.g., SORA [36]) from prompts, and comprehending and automatically suggesting code (e.g., Codex [7]).

Based on the concept of general-purpose programming model Codex, in June 2022, GitHub and OpenAI launched GitHub Copilot to the public as a plug-in to the IDEs, such as Visual Studio Code, Neovim, and JetBrains. Copilot is an artificial intelligence tool that can automatically generate source code from natural language problem descriptions. While both these LLMs are geared towards developers, Copilot is more ideal for developers working on GitHub repositories, as it has been trained on a vast amount of code from GitHub public repositories and has strong GitHub ecosystem support. The Copilot extension offers several key features for multiple programming languages, such as code completion, translating code from one programming language to another, generating comments and documentation, etc. This thesis focuses on exploring the first feature, code completion, which is triggered when a user provides query context. We refer to the combination of function name, function parameters, and/or type annotations, usage context, as the **query context**.

In a 2022 article, Moroz et al. [28] generally examined and summarized the potential and the remained problems of Copilot. The first problem is the quality of Copilot training data. The training data contains unmaintained, legacy, and low-quality code, such as code from novice developers, code with poor style or inefficient algorithms, code that cannot run or compile, and code with deprecated uses of libraries and language syntax. The second problem is that Copilot can only understand limited context and cannot detect different dependencies within the project. Another problem is the over-reliance of novice developers, who make up the majority of Copilot's target audience. And due to the non-deterministic nature of the AI model, Copilot's suggestions are irreproducible. Copilot is also exposed to safety issues, copyright problems, and harmful content.

In another paper, Nguyen and Nadi [29] compared the accuracy and complexity of Copilot's suggested solutions across four programming languages: Python, Java, JavaScript,

and C. After examining 33 LeetCode problems, they found that Java suggestions have the highest accuracy (57%) and JavaScript has the lowest (27%). They used SonarQube [41] to quantify complexity by measuring the cyclomatic and cognitive complexity of Copilot's suggestions. The results showed that there were minuscule variations in understandability amongst the four programming languages, and all suggestions were of low complexity. The study also pointed out some notable shortcomings of Copilot, such as the fact that the proposed code can sometimes be further simplified or relies on undefined helper methods.

Vaithilingam et al. [44] conducted a within-subjects study with 24 participants to understand how developers use and perceive Copilot. Participants were asked to complete three real-world Python programming tasks with different levels of difficulty: editing CSV (easy), web scraping (medium), and graph plotting (hard). The study discovered that, even though Copilot did not improve task completion time or success rate, it did provide a useful starting point for the users, even if the generated code was incorrect. Nevertheless, since Copilot only suggested one code completion option, developers missed out on the opportunity to compare different sources and community discussions in comparison to searching for code on the internet. The study also observed that participants had difficulty comprehending and troubleshooting the code generated by Copilot.

In contrast to the study of Vaithilingam et al. [44], in which participants have diverse experiences with programming, Denny et al. [12] emphasized the influence of Copilot on students. The article presented the first exploration of the efficacy of query context for Copilot in an introductory programming environment. CodeCheck [10] is a public website that provides introductory programming problems with description comments, function names, and input parameters for students to practice by themselves. In this experiment, Copilot was asked to predict the function body for 166 Python problems in CodeCheck. If the prediction failed any test case, no other changes were permitted; only natural language changes to the description comments were permitted in order to further clarify the problem. The results showed that Copilot can successfully solve around half of these problems on its first attempt and 60% of the remaining problems with only natural language changes to the input queries.

Wermelinger [45] also focused on the impact of Copilot on software engineering pedagogy and reported his experience in using Copilot as he portrayed the perspective of a student. This study compared the quality and variety of generated code, tests, and explanations of Copilot to Codex and answered the question, "If the suggestion is incorrect, can we lead Copilot to the correct one?". Similar to other studies, he discovered that Copilot solutions are not very reliable, and the model cannot rectify its answers unless the queries are specifically modified.

Since the official launch of Copilot, there have been a variety of publications examining the model's potential and limitations. Despite the fact that Copilot is very sensitive to the input query context, only few studies have systematically investigated the connection between the modified query context and suggested code quality. To the best of our knowledge, our study is the first to explicitly evaluate the implications of type annotations and usage context on Copilot's predictions.

<pre> 1  def power(x, n): 2      if n == 0: 3          return 1 4      else: 5          return x * power(x, n - 1) </pre>	<pre> 1  def power(x: int, n: int) -&gt; int: 2      if n == 0: 3          return 1 4      else: 5          return x * power(x, n - 1) </pre>
---	---

Figure 2.3: Python code snippet without type annotations (left) and with type annotations (right)

## 2.3 Data Type and Type Annotations

A data type is the characteristic of a variable that determines what kind of data it holds within the program. There are two main types of typing in programming: static typing and dynamic typing. Statically typed languages, for example, C, C++, and Java, require developers to explicitly declare the data type for each variable. These types of languages enforce type checking already at compile time. This allows code completion tools to make predictions without executing the code by leveraging the provided type information. In dynamically typed languages, such as Python and JavaScript, variables are bound to data types at runtime instead of during compilation. Dynamically typed languages have become increasingly popular in recent years [42]. Their flexibility enables concise code and ease of use, as developers do not have to specify every data type. Naturally, the absence of type annotation comes with the trade-offs of program safety, difficulty in maintenance, and limiting autocompletion support from IDEs [17, 32].

A type annotation is a notation used for explicitly defining the type of variables, function parameters, or return variables. Type annotations are prominent in statically typed languages and are becoming more common in dynamically typed languages through mechanisms such as type hints. Figure 2.3 shows an example of a Python code snippet without type annotations on the left and one with type annotations on the right. Type annotations can help to improve code readability, documentation, and support from tools, resulting in more resilient, maintainable, and error-free code [17, 32].

Di Grazia and Pradel [13] published a large-scale analysis of 1,414,936 type annotation changes in Python to understand the characteristics and evolution trend of type annotations and type errors over time. The paper highlighted several intriguing findings. The first finding is that, despite becoming increasingly popular, type annotations in Python are still far from being the norm. However, this trend is gradually shifting in favor of more annotations. The second finding is that type annotation is used regularly in projects with more contributors, while it is used only occasionally in projects with fewer contributors. The last finding is that the number of type annotations added correlates positively with the number of detected type errors. Nevertheless, 78.3% of the commits were still committed despite having errors. This study concluded that adding type annotations is a long-term investment because they are rarely modified later on. The fact that more than 90% of program elements were not yet annotated highlights the need for tools that can infer and anticipate data types.

In recent years, multiple approaches have been proposed to assist developers in managing type annotations in dynamically typed languages. There are three basic techniques for predicting data types in dynamically typed languages. The first technique is static type

```
1  def power(x, n):  
2      if n == 0:  
3          return 1  
4      else:  
5          return x * power(x, n - 1)  
6  myPower = power(2, 3)
```

Figure 2.4: Usage context in Python

inference [3, 14, 21], which relies on abstract interpretation or type constraints to propagate type annotations at compile time. While it can detect type errors early on, the dynamic nature of languages such as JavaScript and Python limits its applicability. The second technique is dynamic type inference [2, 35], which tracks data flows to determine the types of expressions, variables, and functions at runtime. This approach yields precise data types, but errors can be more difficult to trace since they occur during execution. The third technique is probabilistic type prediction, which infers type annotations leveraging statistical methods [46] or machine learning [1, 18, 26, 30, 34] based on patterns observed within the project and external open source code. Copilot utilizes this third technique to predict type annotations, and one of our study goals is to examine how accurate these predictions are.

## 2.4 Usage Context

Usage context in programming consists of the surrounding code environment in which the function or software component is called and used. Figure 2.4 shows an example of usage context, where function `power` is called and consumed by variable `myPower`. According to several studies [37–39], developers spent more time looking up definitions of functions and their uses than variables, statements, or code fragments. As the size of a software project expands, it requires more collaboration among developers. This makes developers often struggle to understand huge codebases, and usage context is usually used as a starting point to understand the role of each function in a program. McMillan et al. [27] thereby presented Portfolio, a code search engine that provides assistance in the discovery of relevant functions and how those functions are used by retrieving and visualizing relevant functions and their applications. While usage context is vital for developers, it is unclear whether it is similarly valuable for LLMs. In addition to type annotation, our study also investigates how usage context influences Copilot’s predictions.

# Methodology

---

In this chapter, we describe our research questions, the materials required, the experiment variables, and the experiment procedure.

## 3.1 Research Questions

Our study investigated the influence of type annotation and usage context on Copilot's predictions by addressing two following questions:

**RQ<sub>1</sub>** How do type annotation and usage context affect Copilot's prediction in terms of **accuracy**?

Despite the fact that [IDEs](#) make heavy use of compile-time type information to provide predictions [\[43\]](#), there is no exact measurement regarding how much Copilot's predictions vary when being provided with type information compared to when not. Moreover, many studies have proven the value of usage context in navigating developers through large code-bases [\[37–39\]](#). While most developers rely on usage context to understand the behavior of a function, is it similarly important to Copilot? Our study investigated how type annotation and usage context affect the accuracy of Copilot when predicting function bodies.

**RQ<sub>2</sub>** How do type annotation and usage context affect Copilot's prediction in terms of **complexity**?

Code complexity is a critical aspect of software development. Developers are responsible for not only developing software but also ensuring that it is reliable, optimal, and easy to maintain over time. In addition to accuracy, we also investigated how type annotation and usage context affect the complexity of function bodies predicted by Copilot.

## 3.2 Materials

### 3.2.1 Programming Language: Python

We conducted our tests on dynamically typed languages rather than statically typed languages, as type checking in dynamically typed languages occurs only during program

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
```

Figure 3.1: *TreeNode* class

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
```

Figure 3.2: *ListNode* class

execution, and our input queries remain syntactically correct even without type annotations. We selected Python over other dynamically typed languages because it is the most popular and widely used general-purpose programming language, particularly in data science and machine learning [42]. Another reason is that Codex, an ancestor model of Copilot, has been found to be most capable in Python [7]. And Moroz et al. [28] found that Copilot demonstrated remarkable results while working with Python, maybe because Python is easily readable and comparable to human language.

Python consists of multiple data types, such as: numeric data types (int, float, complex), string data types (str), sequence types (list, tuple, range), binary types (bytes, bytearray, memoryview), mapping data type (dict), boolean type (bool), and set data types (set, frozenset). However, in our study we only evaluated commonly used and fundamental Python data types int, float, str, list, and two custom data types *TreeNode* (cf. Figure 3.1) and *ListNode* (cf. Figure 3.2) from LeetCode.

### 3.2.2 Programming Problems: CodeCheck and LeetCode

To test Copilot’s autocompletion ability, we need a balanced, sufficiently large, and ideally unbiased set of programming problems. Horstmann’s CodeCheck [10] and LeetCode [25] are two public online collections that meet our requirement. While CodeCheck was designed for novice developers, LeetCode focuses more on experienced developers. Each problem from these two collections also comes with test cases and the relevant information to compose comprehensive query contexts for Copilot, such as problem descriptions, function names, number of input and return parameters, and type annotations of input and return parameters of the function. More importantly, CodeCheck and LeetCode were used to test Copilot’s autocompletion ability in previous studies by Denny et al. [12] and Nguyen and Nadi [29]. As Copilot’s suggestions are nondeterministic, it is valuable to investigate the same programming problems so that future studies can use our findings and compare them to prior works to determine whether and how Copilot’s predictions have changed in the course of time.

#### 3.2.2.1 CodeCheck

Horstmann’s CodeCheck [10] is an autograder website that provides a large collection of publicly accessible common programming problems. This website aims to help instructors provide self-practice exercises for their students [23]. Each of the CodeCheck problems comprises the information needed to create a comprehensive query context as well as test cases to evaluate the accuracy of Copilot’s completions.



We randomly selected 50 programming problems with several criteria (discussed in Section 3.4) from the Python test bank on CodeCheck [9]. The 50 problems are divided into four main categories:

- **Branches:** These problems required some combination of `if/elif/else` statements. We selected 7 problems from this category.
- **Strings:** These problems required the use of loops (over the characters of an input string), string slicing, indexing, and basic string methods (e.g., `isdigit()`, `split()`), but without lists or other data structures. We selected 9 problems from this category.
- **Lists:** These problems involved searching through lists, counting, averaging, adding/removing/swapping elements, etc. We selected 20 problems from this category.
- **Two-Dimensional Arrays:** These problems involved one-dimensional (list) and two-dimensional (list of lists) arrays and required processing some combination of all elements, or corners, borders and diagonals. We selected 14 problems from this category.

Consider the programming problem `swapFirstAndLast` as shown in Figure 3.3, lines 2 to 4 contain the natural language problem description, parameter type, and return type of the function. On line 6, CodeCheck provides the function name `swapFirstAndLast` and the input parameter (`s`). All this information corresponds to the attributes that compose a comprehensive query context for Copilot.

CodeCheck's coding environment also contains a set of predefined test cases, allowing us to measure the behavioral accuracy of Copilot's suggestions. After pasting the function body suggested by Copilot in place of the comment `# Your code here...` and pressing CodeCheck button, CodeCheck returns the submission statuses for each test case and the overall number of test cases passed, as shown in Figure 3.4. To be consistent with the submission statuses in LeetCode, we define three possible statuses for CodeCheck:

- **Accepted:** submitted code passes all test cases.
- **Wrong Answer:** submitted code has no errors, but its output is different from the expected output for at least one test case.
- **Runtime Error:** submitted code has at least one test case that fails due to errors during execution (e.g., division by zero).

### 3.2.2.2 LeetCode

LeetCode [25] is a popular online platform for coding interview preparation and competitions that contains a large number of programming and algorithmic problems. Each of the LeetCode problems also comprises the information needed to create a comprehensive query context as well as test cases to evaluate the accuracy of Copilot's completions.

We randomly selected 50 programming problems from the set of all problems on LeetCode [24]. The 50 problems are divided into three difficulty levels:

prog.py

```

1
2 # Given a string, return the string with the first and last characters swapped. If
3 # there are fewer than 2 characters in the string, return the original string. Do not
4 # use a loop to solve this problem.
5
6 def swapFirstAndLast(s):
7     # Your code here...

```

CodeCheck

Reset

Figure 3.3: CodeCheck coding environment

prog.py

```

1
2 # Given a string, return the string with the first and last characters swapped. If
3 # there are fewer than 2 characters in the string, return the original string. Do not
4 # use a loop to solve this problem.
5
6 def swapFirstAndLast(input):
7     return input[-1] + input[1:-1] + input[0]

```

CodeCheck

Reset

Download

Calling with Arguments

	Name	Arguments	Actual	Expected
pass	swapFirstAndLast	"cba"	abc	abc
pass	swapFirstAndLast	"oellh"	hello	hello
fail	swapFirstAndLast	"a"	aa	a
pass	swapFirstAndLast	"ba"	ab	ab
fail	swapFirstAndLast	""	Traceback (most recent call last): File "progCodeCheck.py", line 19, in <module> main() File "progCodeCheck.py", line 17, in main result = prog.swapFirstAndLast("") File "/tmp/codecheck/workCaKXSXKkM2CIfqkKvMhk/submissioncall/prog.py", line 7, in swapFirstAndLast return input[-1] + input[1:-1] + input[0] IndexError: string index out of range	

Score

3/5

Figure 3.4: CodeCheck submission status



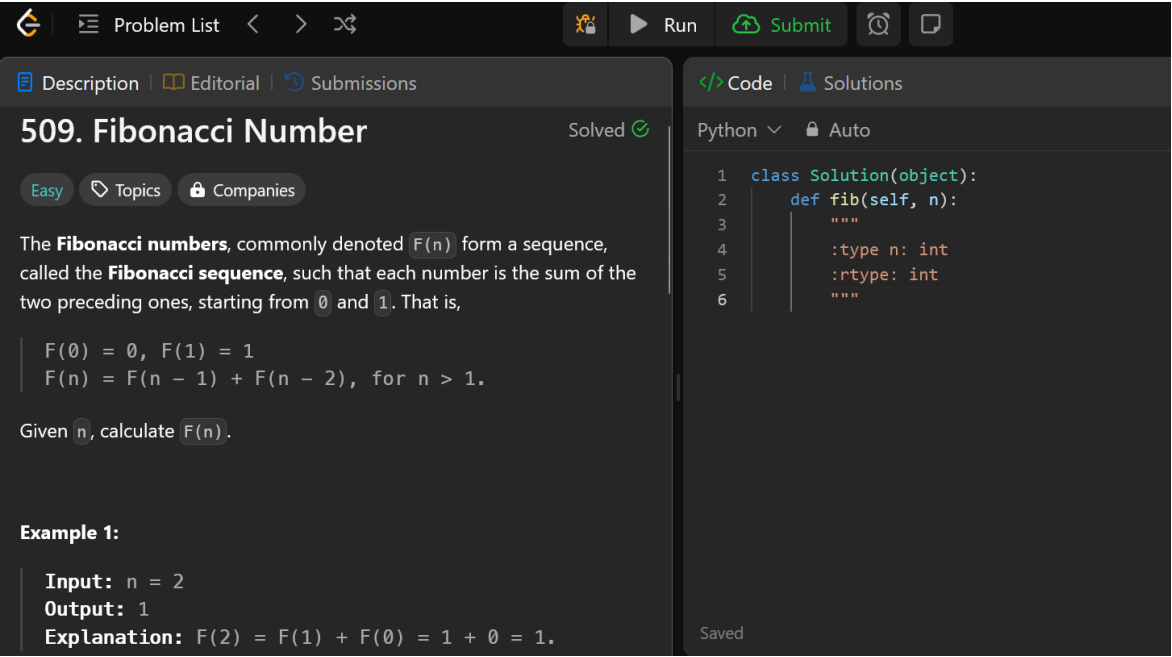


Figure 3.5: LeetCode coding environment

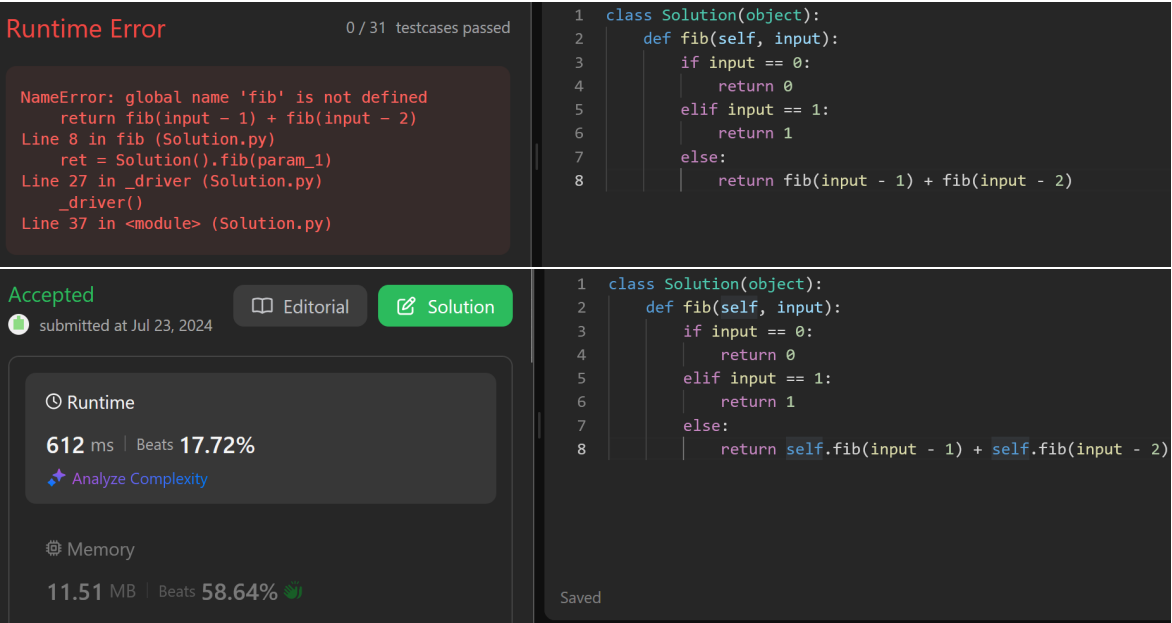


Figure 3.6: LeetCode submission statuses

- **Easy:** These problems have straightforward solutions and only test basic data structures and algorithms. We selected 17 problems from this category.
- **Medium:** These problems either test more advanced data structures and algorithms, or basic data structures and algorithms in more complex situations. We selected 21 problems from this category.
- **Hard:** These problems either test the most advanced data structures and algorithms within three levels, or they test more obscure and rarely encountered algorithms, mathematics, and data structures. We selected 12 problems from this category.

Consider a common programming problem, Fibonacci Number, as shown in Figure 3.5. As LeetCode supports multiple programming languages, we must first manually set the programming language in the LeetCode coding environment to Python. The left window contains the natural language problem description and examples of the problem. On line 2 of the right window, LeetCode provides the function name `fib` and the parameter (`self`, `n`). Because Python does not require type annotations, parameter type and return type are provided as comments from line 3 to line 6.

LeetCode's coding environment also contains a set of predefined test cases. After pasting the function body suggested by Copilot in place of the comment and pressing Submit button, LeetCode returns the submission status and the overall number of test cases passed, as shown in Figure 3.6. There are six possible statuses:

- **Accepted:** submitted code passes all test cases.
- **Wrong Answer:** submitted code has no errors, but its output is different from the expected output for at least one test case.
- **Time Limit Exceed:** submitted code has no errors, but at least one test case exceeds permitted execution time.
- **Memory Limit Exceed:** submitted code has no errors, but at least one test case exceeds permitted execution memory.
- **Compile Error:** submitted code cannot be compiled (Python is not a compiled language, so its code will not result in any compilation error).
- **Runtime Error:** submitted code has at least one test case that fails due to errors during execution (e.g., division by zero).

When one test case fails or causes a runtime error, CodeCheck continues executing other test cases and counts the total number of test cases passed, see Figure 3.4. However, LeetCode halts execution after the first failed test and reports the number of test cases passed without executing the remaining. When one test case causes a runtime error, LeetCode immediately halts execution, indicates the number of tests passed as 0, as shown in Figure 3.6. For that reason, we provided a lower bound for the number of test cases passed in some cases.

## 3.3 Variables

### 3.3.1 Independent Variables: Type Annotation and Usage Context

We selected two independent variables, type annotation and usage context, based on their relevance to the theoretical framework guiding this study and their demonstrated impact in related works. Previous studies have shown that type annotation helps improve IDEs' support [13, 17, 32]. Usage context represents the surrounding code environment in which the function is invoked, and Copilot makes predictions based on these surrounding code. By analyzing type annotation and usage context, we seek to provide a comprehensive understanding of how much the presence of these two independent variables affects Copilot's autocompletion ability, offering insights that can contribute to both theoretical advancements and practical applications. In our experiment, we compare the four associated function bodies generated by Copilot when given four different types of query contexts:

- **No type annotation, no usage context (NONE):** This type of query only contains the function name and the number of input parameters. For instance, the query context in Figure 3.7 provides information about the function name `fib` and the number of input parameters (`input`).
- **Type annotation, no usage context (TA):** This type of query contains the function name, the number of input and return parameters, and type annotations of input and return parameters. For instance, the query context in Figure 3.8 provides information about the function name `fib`, the number and type annotations of input parameters (`input: int`), and the number and type annotation of return parameters `-> int`.
- **No type annotation, usage context (CON):** This type of query contains the function name, the number of input parameters, and an example usage context of the function. For instance, the query context in Figure 3.9 provides information about the function name `fib`, the number of input parameters (`input`), and an example usage context `s = fib(2)`.
- **Type annotation, usage context (TA\_CON):** This type of query contains the function name, the number of input and return parameters, type annotations of input and return parameters, and an example usage context of the function. For instance, the query context in Figure 3.10 provides information about the function name `fib`, the number and type annotations of input parameters (`input: int`), the number and type annotation of return parameters `-> int`, and an example usage context with return type annotation `s: int = fib(2)`.

We made two modifications to the initially provided function parameters. The first modification was to replace all initial parameter names with `input` to prevent parameter names from revealing data types and affecting Copilot's predictions. The second modification was to remove `self` from LeetCode functions' parameters. For example, the LeetCode

```
def fib(input):
```

Figure 3.7: **NONE**: No type annotation, no usage context query

```
s = fib(2)
def fib(input):
```

Figure 3.9: **CON**: No type annotation, usage context query

```
def fib(input: int) -> int:
```

Figure 3.8: **TA**: Type annotation, no usage context query

```
s: int = fib(2)
def fib(input: int) -> int:
```

Figure 3.10: **TA\_CON**: Type annotation, usage context query

function `def fib (self, n):` will have the **NONE** query `def fib(input):`. If we directly paste Copilot’s suggested function bodies that contain recursive calls into the LeetCode coding environment, it would create runtime errors. Due to that reason, before submitting Copilot’s suggestion, we need to explicitly add `self.` in front of each recursive call, as seen in Figure 3.6.

### 3.3.2 Dependent Variables: Accuracy and Complexity

In this study, two dependent variables are the accuracy and complexity of Copilot’s suggestions, which will be discussed in more detail next.

#### 3.3.2.1 RQ1: Accuracy

Accuracy is the primary prerequisite for producing high-quality code, demonstrating how trustworthy it is. As Copilot is increasingly employed in the software engineering industry, it is critical that it can generate accurate code, which contributes to the software’s reliability, security, performance, and efficiency. For accuracy, we employed LeetCode existing test cases, which ensure that Copilot’s suggestions adhere to various time and space constraints and pass corner cases for the given problem, as well as CodeCheck test cases, which simply validate whether the suggestions produce the expected output. All possible submission statuses were addressed in previous Subsection 3.2.2 (Programming Problems). It should be noted that the accuracy of some LeetCode cases is only the lower bound because LeetCode stops executing the remaining tests when one fails, or stops executing then reports zero test passed when one causes a runtime error.

#### 3.3.2.2 RQ2: Complexity

A secondary prerequisite for high-quality code is complexity, indicating how easy the code is for developers to test and understand. Because developers are responsible for not just developing software but also maintaining it throughout time. We used SonarQube [41] to measure the cyclomatic and cognitive complexity of Copilot’s suggestions. SonarQube is an open-source platform that statically analyzes code and reports on coding standards, unit tests, code coverage, code complexity, and security recommendations.

Cyclomatic complexity measures the number of distinct paths through the code, indicating how complex the logic is. A lower cyclomatic complexity implies simpler, more manageable

```

1  def fib(input):
    if input == 0:
        return 0
    elif input == 1:
        return 1
    else:
        return fib(input - 1) + fib(input - 2)

```

Figure 3.11: Example of Copilot’s code completion

code, which reduces the possibility of errors and makes it easier to maintain and modify. A higher cyclomatic complexity implies more branching in the code and more test cases required to fully cover a function [6]. To calculate cyclomatic complexity, SonarQube starts with a value of one and increments by one whenever it detects a split in the function control flow that creates a new conditional branch [40]. Nevertheless, we cannot estimate the cyclomatic complexity of Python code that is syntactically invalid.

In contrast to cyclomatic complexity, which refers to how difficult the code is to test, cognitive complexity indicates how difficult the code is to mentally process and understand. Rather than mathematical models that analyze control flow, SonarQube leverages rules that map into a programmer’s intuitive understanding of code [5]. To measure cognitive complexity, SonarQube does not increment the complexity score when shorthands are used (e.g., using a ternary expression), it increments the score only once for each break in the code’s linear flow (e.g., a whole switch statement only increments the score by one as it can often be understood with one scan), and increments the score when flow-breaking structures are nested.

## 3.4 Procedure

In this section, we explain four main steps of our experiment.

### Step 1: Gather programming problems and generate query contexts

We went through each category of CodeCheck’s Python test bank [9] and each difficulty level of LeetCode’s problem set [24] to arbitrarily select 100 programming problems, 50 problems per website. To be eligible, the problem must not be longer than 30 lines, not reveal any data type information in its function name, and contain common Python data types, such as `int`, `float`, `str`, `list`. We also selected one problem `hasPathSum` that contains data type `TreeNode` (cf. Figure 3.1) and two problems `removeNthFromEnd`, `swapPairs` that contain data type `ListNode` (cf. Figure 3.2) from LeetCode.

From the problem description on CodeCheck and LeetCode, for each problem, we manually extracted information about function name, the number of input and return parameters, type annotations of input and return parameters, and an example usage context to create four query contexts `NONE`, `TA`, `CON`, `TA_CON` (cf. Figures 3.7, 3.8, 3.9, 3.10). Note

that we did not take into account the parameter `self` of LeetCode problems. In total, we generated 400 query contexts for 100 programming problems.

## Step 2: Acquire Copilot's completions.

Applying for GitHub Education [16] offers us free access to Copilot, which was then installed as a plug-in in IDEs Visual Studio Code. The Copilot version we used in this study is v1.194.0. We first entered queries from Step 1 into the Visual Studio Code editor, manually invoked Copilot for each query, waited for Copilot to complete the function body, accepted it by pressing Tab key, and saved the suggested completions in separate Python files. Figure 3.11 depicts an example of using Copilot. After creating a new Python file on Visual Studio Code and entering query context `def fib(input):`, the suggested completion appears after the cursor position, below the query, and can be accepted using the Tab key. Since Copilot is non-deterministic and usually produces different suggestions, we only evaluated its first suggested completion. Overall, we obtained 400 Python files corresponding to 400 query contexts.

## Step 3: Evaluate accuracy.

Figures 3.12 and 3.13 depict how to include Copilot's suggested completions into CodeCheck's and LeetCode's coding environments. We first need to change the initial function parameter names to `input`, then we pasted Copilot's completions in place of the comment `# Your code here...` for CodeCheck coding environment or in place of the data types comment from line 3 to line 6 for LeetCode coding environment. If the LeetCode suggested function body contains recursive calls, we manually append `self.` before each call. After that, we ran Copilot's suggested completions against test cases and reported the submission statuses. The average number of test cases for the 50 CodeCheck selected problems is 4.78 and for the 50 LeetCode selected problems is 345. Tables A.1 and A.2 summarize the accuracy of 400 Copilot's completions, including the number of tests passed and the submission statuses.

## Step 4: Evaluate complexity.

In addition to accuracy, we also assessed the cyclomatic and cognitive complexity of Copilot's completions. We ran SonarQube [41] on 400 Python files containing 400 Copilot's completions, then summarized the measured cyclomatic complexity on two plot boxes in Figures 4.6, 4.7 and the measured cognitive complexity on two plot boxes in Figures 4.8, 4.9. We did not remove type annotations and usage contexts from Python files while running SonarQube, since they did not increase the function complexity. Note that we cannot measure the cyclomatic complexity of syntactically invalid Python code (6 cases in LeetCode and 1 case in CodeCheck).

```

6 def swapFirstAndLast(s):
7     # Your code here...

```

↓

```

6 def swapFirstAndLast(input):
7     # Your code here...

```

↓

```

6 def swapFirstAndLast(input):
7     if len(input) < 2:
8         return input
9     else:
10        return input[-1] + input[1:-1] + input[0]

```

Figure 3.12: Modify CodeCheck's coding environment to measure accuracy

```

1 class Solution(object):
2     def fib(self, n):
3         """
4         :type n: int
5         :rtype: int
6         """

```

↓

```

1 class Solution(object):
2     def fib(self, input):
3         """
4         :type n: int
5         :rtype: int
6         """

```

↓

```

1 class Solution(object):
2     def fib(self, input):
3         if input == 0:
4             return 0
5         elif input == 1:
6             return 1
7         else:
8             return fib(input - 1) + fib(input - 2)

```

↓

```

1 class Solution(object):
2     def fib(self, input):
3         if input == 0:
4             return 0
5         elif input == 1:
6             return 1
7         else:
8             return self.fib(input - 1) + self.fib(input - 2)

```

Figure 3.13: Modify LeetCode's coding environment to measure accuracy





# Results

In this chapter, we present a summary of the most important results on Copilot's completion accuracy and complexity. This summary provides a segue into Chapter 5 (Discussion), where these results will be discussed in detail.

## 4.1 RQ1: Accuracy

When evaluating the quality of LLMs such as Copilot, the primary and foremost criteria to take into account is the accuracy of the models' predictions. As software grows in size and complexity, making it impossible for humans to write code from scratch, Copilot has a significant impact and contribution to the software engineering industry. A highly accurate output is critical to reducing potential errors and encouraging more developers to rely on Copilot, leading to more reliable and robust software.

Overall, CodeCheck and LeetCode have completely different accuracy tendencies. In CodeCheck, the order of least to most accurate queries is **NONE**, **CON**, **TA**, **TA\_CON**, but in LeetCode it is **NONE**, **CON**, **TA\_CON**, **TA**. Copilot's suggestions for CodeCheck have average accuracy and vary substantially across four query contexts. LeetCode, on the other hand, has exceptionally high accuracy with little to no variation over four queries.

In particular, for CodeCheck, **TA\_CON** query triggered the most accurate completions, with 72.8% of total test cases passed and 25/50 problems accepted. **TA** query also has 25/50 acceptances but passes fewer test cases (67.8%). The absence of type annotation and usage context in **NONE** query results in the least accurate completions, with only 41.4% of total test cases passed and 15 problems accepted. It also causes the most runtime errors, with 12 problems, which is substantially higher than the other three queries (4, 2, and 3 problems). The gap between the most accurate and the least accurate query contexts is 31.4% of tests passed and 10 accepted problems. We calculated the correlations of the number of passed/failed tests and four query contexts for a total of six test-hypothesis pairs: **NONE** vs. **TA**, **NONE** vs. **CON**, **NONE** vs. **TA\_CON**, **TA** vs. **CON**, **TA** vs. **TA\_CON**, **CON** vs. **TA\_CON**. Since six hypotheses were simultaneously tested, the multiple comparisons problem arises [20]. To account for this problem, we applied a Bonferroni correction and used  $\rho < 0.05/6$  as the threshold of statistical significance [8]. The Chi-square test showed that four queries have a statistically significant influence on the observed passed/failed tests ( $\chi^2(2) = 57.20$ ,  $\rho < 0.001$ ). We provided a tabular view of each CodeCheck programming problem with its accuracy rate in Table A.1.

On the contrary, LeetCode has exceptionally high accuracy, with above 90% of the test cases passed across all four query contexts. In LeetCode, **TA** query triggers the most accurate

Table 4.1: Frequency of patterns in Copilot’s predictions

Pattern	Number of problems	
	CodeCheck	LeetCode
Two no-type-annotation queries ( <b>NONE</b> , <b>CON</b> ) or two type-annotation queries ( <b>TA</b> , <b>TA_CON</b> ) trigger similar completions	10	5
Two no-usage-context queries ( <b>NONE</b> , <b>TA</b> ) or two usage-context queries ( <b>CON</b> , <b>TA_CON</b> ) trigger similar completions	6	2
No type annotation, no usage context query ( <b>NONE</b> ) triggers type-error completion	12	4
Copilot’s completions call helper function	1	6

completions, with 95.5% of tests passed and 37/50 problems accepted. The least accurate query is still **NONE** query, with 91.3% of tests passed and 31 problems accepted. Intriguingly, the gap between the most accurate and least accurate query is very small, only around 4% and 6 problems difference in the number of total tests passed and accepted completions, respectively. We also applied a Bonferroni correction, setting  $\rho < 0.05/6$  as the threshold of statistical significance for six test-hypothesis pairs and found statistically significant differences between four query contexts ( $\chi^2(2) = 268.08, \rho < 0.001$ ). We provided a tabular view of each LeetCode programming problem with its accuracy rate in Table A.2.

When comparing **TA** queries containing only type annotation information with **CON** queries containing only usage context information, we noticed that type annotation helps Copilot predict more accurately than usage context by approximately 10% for CodeCheck and 2% for LeetCode.

After analyzing 400 Copilot’s completions, we discovered several intriguing patterns, which we categorized and statistically reported in Table 4.1.

- **Two no-type-annotation queries (**NONE**, **CON**) or two type-annotation queries (**TA**, **TA\_CON**) trigger similar completions:** This pattern examines the impact of type annotations on Copilot’s completions by answering two questions: Are the two completions triggered by **NONE** and **CON** queries similar? Are the two completions triggered by **TA** and **TA\_CON** queries similar? We excluded cases in which all four queries triggered the same completions and only included cases that answered true for at least one out of the two questions above. We found 10 CodeCheck (one-fifth of the CodeCheck testing dataset) and 5 LeetCode problems that have identical completions within the same type annotation query.
- **Two no-usage-context queries (**NONE**, **TA**) or two usage-context queries (**CON**, **TA\_CON**) trigger similar completions:** This pattern examines the impact of usage context on Copilot’s completions by answering two questions: Are the two completions triggered by **NONE** and **TA** queries similar? Are the two completions triggered by **CON** and **TA\_CON** similar? We also excluded cases in which all four queries triggered the same completions and only included cases that answered true for at least one out of the two questions above. This pattern appears in only 6 CodeCheck and 2 LeetCode problems, less than the above type annotation patterns.

<pre> 1 def splitEvenOddIndices(input): 2     even = [] 3     odd = [] 4     for i in range(len(input)): 5         if i % 2 == 0: 6             even.append(input[i]) 7         else: 8             odd.append(input[i]) 9     return even, odd </pre>	<pre> 1 def splitEvenOddIndices(input: List[int]) -&gt; List[List[int]]: 2     even = [] 3     odd = [] 4     for i in range(len(input)): 5         if i % 2 == 0: 6             even.append(input[i]) 7         else: 8             odd.append(input[i]) 9     return [even, odd] </pre>
--	---

Figure 4.1: Similar Copilot’s completions for **NONE** and **TA**, only return parameters are different

- **No type annotation, no usage context query (**NONE**) triggers type-error completion:**

This pattern considers cases in which queries with no information about type annotation or usage context trigger Python `TypeError` and cause runtime errors. This pattern appears in 12 CodeCheck and 4 LeetCode problems. The most common `TypeError`s are calling unsupported methods in incorrect data types (e.g., call `len()` in `int` data type), concatenating or mathematically comparing different data types, and trying to interpret list object as an integer.

Interestingly, we found 3 CodeCheck problems in which all four queries generate correct function bodies, but only **NONE** and **CON**, which lack information about type annotations, return incorrect format output. In Figure 4.1, both queries produce similar function bodies. However, the query on the left lacks information about the return type and yields two return values `even, odd`, whereas the correct return parameter should be `[even, odd]` as in the query on the right.

- **Copilot’s completions call helper function:** This pattern considers problems where at least one out of its four suggested function bodies contains additional helper functions. As shown in Figure 4.2, rather than directly implementing the main function `generateParenthesis`, Copilot created and implemented a new auxiliary function `backtrack`. We found 1 CodeCheck and 6 LeetCode problems have this pattern, and most of the helper function names are `backtrack`. In a previous empirical study on Copilot, Nguyen and Nadi [29] observed the same pattern in which Copilot’s suggested function body included a helper function and its implementation. The authors contacted Copilot’s team and confirmed that Copilot’s suggestions should not include the implementation of helper functions, and this is an unexpected Copilot behavior.

Aside from the above patterns, we noticed three types of `SyntaxError` in Copilot’s suggested function body. The first type of `SyntaxError` is the missing of closing parenthesis (cf. Figure 4.3), which occurred in 1 CodeCheck and 1 LeetCode problem. The second `SyntaxError` type appeared in 2 LeetCode problems where Copilot tried to complete type annotation for no-type-annotation query contexts (cf. line 1 of Figure 4.4). The third `SyntaxError` type is not due to Copilot. Figure 4.5 shows a syntactically correct suggested function body, but when executed on LeetCode, it indicates two syntax errors on line 3 `backtrack(s: str, left: int, right: int):` and line 11 `res: List[str] = []`. Removing type annotations from these two lines resulted in a function body resembling the one in Figure 4.2, which is still syntactically correct Python code and passes all test cases. This

```

1  def generateParenthesis(input: int) -> List[str]:
2      def backtrack(s, left, right):
3          if len(s) == 2 * input:
4              result.append(s)
5              return
6          if left < input:
7              backtrack(s + '(', left + 1, right)
8          if right < left:
9              backtrack(s + ')', left, right + 1)
10     result = []
11     backtrack('', 0, 0)
12     return result

```

Figure 4.2: Copilot's completion contains backtrack helper function

```

1  def findTheDifference(input1, input2):
2      return chr(sum(map(ord, input2)) - sum(map(ord, input1)))

```

Figure 4.3: SyntaxError: miss closing parenthesis

error appeared in 6 **TA\_CON** and 2 **TA** LeetCode queries, reducing the number of tests passed by 3.2% and 0.6%, respectively.

Overall, these results lead us to answer our first research question:

**RQ1**

CodeCheck and LeetCode displayed different accuracy patterns. In CodeCheck, the accuracy of queries from least to most accurate follows this order: **NONE**, **CON**, **TA**, **TA\_CON**, while in LeetCode the order is: **NONE**, **CON**, **TA\_CON**, **TA**. Copilot's suggestions in CodeCheck exhibited average accuracy with significant variation across the four query contexts, whereas in LeetCode, the accuracy is exceptionally high (over 90%) with minuscule variation.

## 4.2 RQ2: Complexity

The second criterion we considered to assess Copilot's prediction is code complexity. Given that Copilot has recently been released as a commercial product and offered free for students, the majority of Copilot's target audiences are novice developers, who are inexperienced and

```

1  def isPalindrome(input): str -> bool:
2      input = input.lower()
3      input = ''.join(e for e in input if e.isalnum())
4      return input == input[::-1]

```

Figure 4.4: SyntaxError: append type annotations on no-type-annotation query

```

1  s: List[str] = generateParenthesis(3)
2  def generateParenthesis(input: int) -> List[str]:
3      def backtrack(s: str, left: int, right: int) -> None:
4          if len(s) == 2 * input:
5              result.append(s)
6              return
7          if left < input:
8              backtrack(s + '(', left + 1, right)
9          if right < left:
10             backtrack(s + ')', left, right + 1)
11     result: List[str] = []
12     backtrack('', 0, 0)
13     return result

```

Figure 4.5: SyntaxError: redundant type annotations

tend to over-rely on Copilot’s suggested code completions [28]. As these suggestions serve as a starting point for novice developers, Copilot should provide code suggestions that are easy to maintain and understand, which means they have low cyclomatic and cognitive complexity.

Copilot generally produces low-complexity code. While the complexity varies between four CodeCheck query contexts, there is no notable variation in LeetCode. Figures 4.6 and 4.7 provide a summary of the cyclomatic complexity, while Figures 4.8 and 4.9 summarize cognitive complexity for 50 CodeCheck and 50 LeetCode programming problems.

The order of highest to lowest complexity queries in CodeCheck is **NONE**, **TA** and **CON**, **TA\_CON**. Consider four queries in CodeCheck, **NONE** query with the least information triggered the most complex completions (cyclomatic and cognitive complexity medians are 3 and 4, respectively), while **TA\_CON** containing the most information triggered the least complex completions (cyclomatic and cognitive complexity medians are 2 and 1.5, respectively). **TA** and **CON** queries both have the same cyclomatic and cognitive complexity median of 3. The gaps between queries with the lowest complexity and the highest complexity in CodeCheck are 1 for cyclomatic and 2.5 for cognitive complexity. However, the complexity of all four LeetCode queries has the same median of 4.

To determine whether the four queries affect Copilot’s completions in terms of complexity, we applied the Kruskal-Wallis test. The results indicated that only the cyclomatic complexity in CodeCheck showed statistical significance with  $\rho = 0.02 < 0.05$ , indicating that the query selection significantly affects the cyclomatic complexity of Copilot’s predictions for CodeCheck problems. The other three cases, CodeCheck cognitive complexity and LeetCode both cyclomatic and cognitive complexity, did not show statistical significance, with  $\rho = 0.06$ ,  $\rho = 0.96$ ,  $\rho = 0.99 > 0.05$ , respectively. As a post-hoc analysis, Dunn’s test with Bonferroni-adjusted  $\rho$ -values was conducted for CodeCheck cyclomatic complexity, which is the only significant result from the Kruskal-Wallis test. The  $\rho$ -values for the six query pairs showed that only the difference between the **NONE** vs. **TA\_CON** queries is statistically significant with  $\rho = 0.03 < 0.05$ .

One pattern we notice in CodeCheck problems is that Copilot usually invokes Python built-in methods (e.g., `sum()`, `min()`, `max()`, `len()`, `sort()`, etc.) rather than implementing

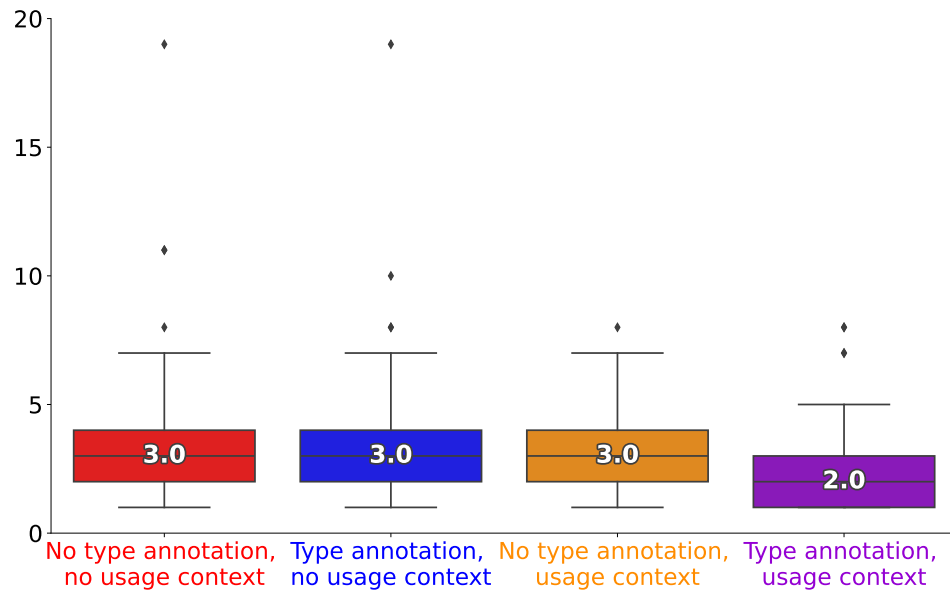


Figure 4.6: Cyclomatic complexity of Copilot's completions for 50 CodeCheck problems

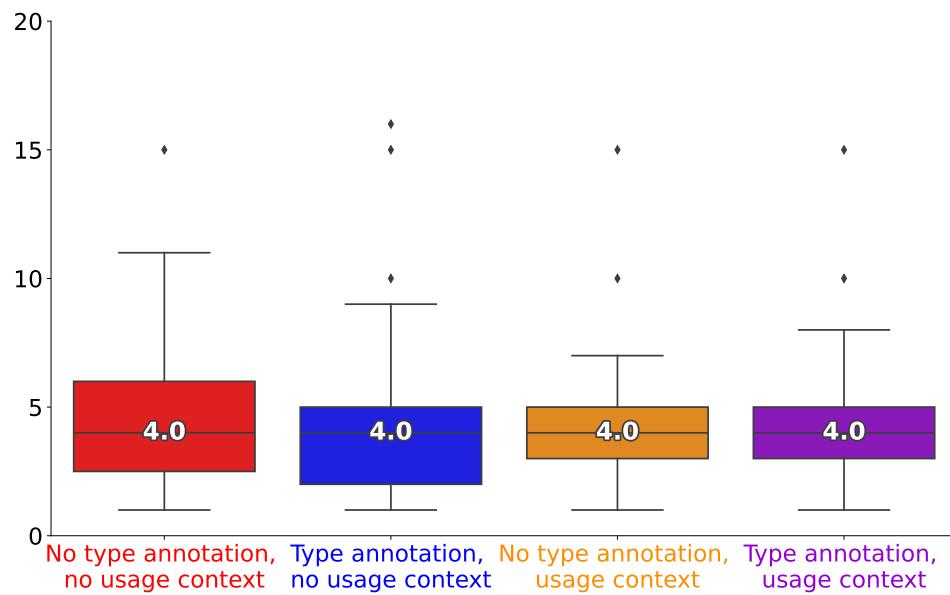


Figure 4.7: Cyclomatic complexity of Copilot's completions for 50 LeetCode problems

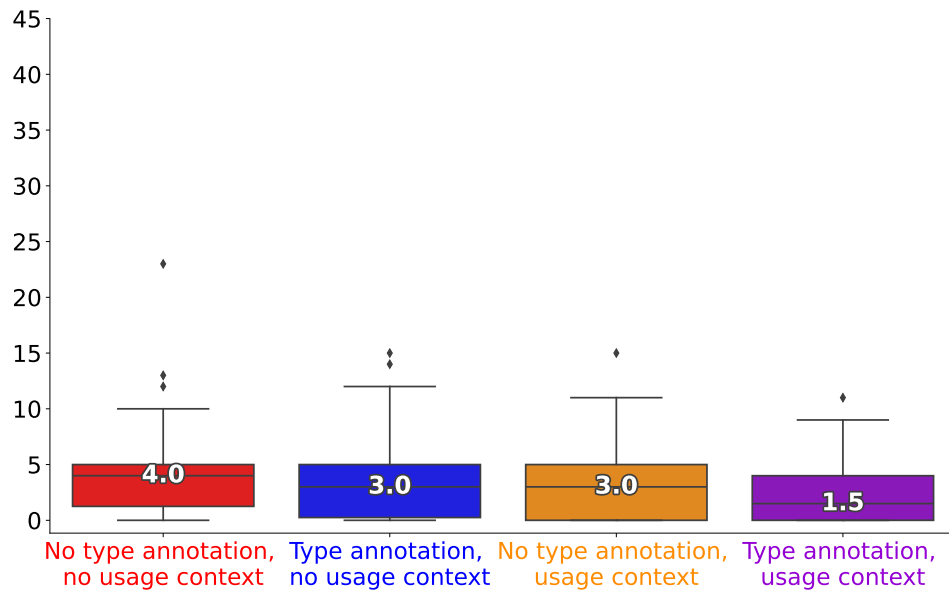


Figure 4.8: Cognitive complexity of Copilot's completions for 50 CodeCheck problems

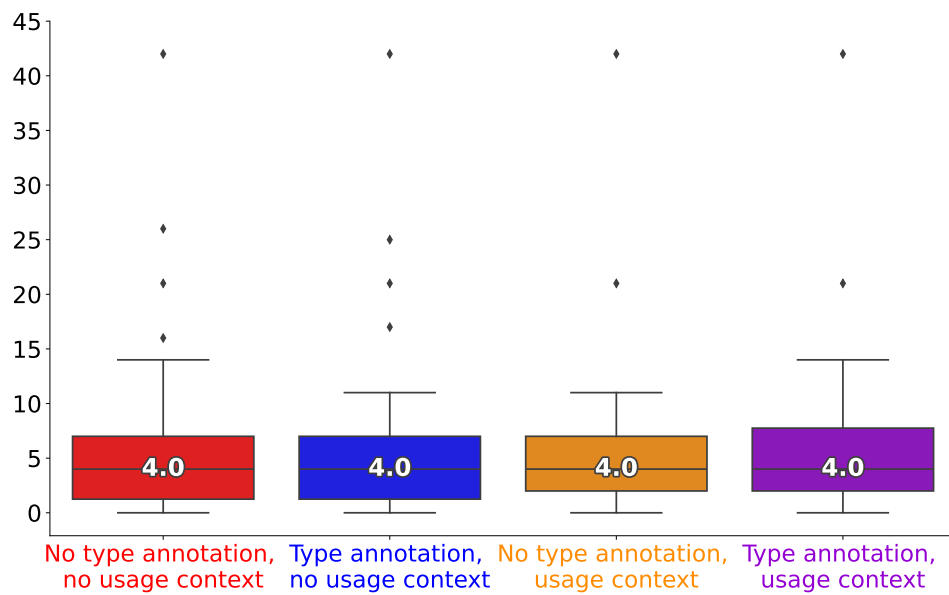


Figure 4.9: Cognitive complexity of Copilot's completions for 50 LeetCode problems

them from scratch, or compacts the function body (e.g., combining both `for-loop` and `if-statement` into one line). This pattern only appears in 8 **TA** queries and in 9 **CON** queries, where 24 **TA\_CON** queries show this pattern, substantially higher compared to the other two. This could explain why **TA\_CON** query resulted in the overall lowest complexity completions.

Overall, these results lead us to answer our second research question:

**RQ2** Copilot generates low-complexity code for our queries. In CodeCheck, the queries tend to follow an order of complexity from highest to lowest: **NONE**, **TA** and **CON**, **TA\_CON**. In LeetCode, the variation is minuscule, with all queries having a median complexity of 4. The statistical test showed that the choice of query only significantly affects the cyclomatic complexity of Copilot's predictions for CodeCheck problems, in which only the difference between the **NONE** vs. **TA\_CON** queries is statistically significant.



## Discussion

---

In this chapter, we address and explain the reasons behind the results in the previous Chapter 4 (Results).

### 5.1 RQ1: Accuracy

As already mentioned, although CodeCheck continues to execute the remaining test cases, LeetCode immediately stops when one test fails or a runtime error occurs, resulting in lower bound accuracy. This did not affect the submission statuses or the overall results, as the total LeetCode tests passed of four queries are all over 90%, substantially higher than CodeCheck. GitHub's internal evaluation of Copilot showed that it achieved 43% accuracy on the first try completing Python function bodies [15], similar to the result of our CodeCheck **NONE** query (41.4% of tests passed).

We anticipate a pattern where the amount of information positively correlates with Copilot's prediction accuracy. The expected order of the least to most accurate queries is **NONE**, **TA** and **CON**, **TA\_CON**, however, CodeCheck order is **NONE**, **CON**, **TA**, **TA\_CON** and LeetCode order is **NONE**, **CON**, **TA\_CON**, **TA**. When comparing the expected and actual accuracy orders, we noticed two differences. The first difference is that type annotation improves Copilot's prediction accuracy more than usage context, with **TA** vs. **CON** in CodeCheck is 67.8% vs. 57.7%, and in LeetCode is 95.5% vs. 93.9%. This could be because type annotation is important in more programming problems than usage context. For example, Copilot can only predict the correct function body when it knows about the function return types (cf. Figure 4.1). However, there are some programming problems that are more biased usage contexts, in which Copilot needs to explicitly know about the usage context for correct predictions. For example, in Figure 5.1, the below completion is correct because it recognizes the input string as "R" instead of 'row' as the above completion. We cannot conclude whether type annotations or usage contexts are more valuable to Copilot. The second difference is that in LeetCode, **TA** has the highest accuracy instead of **TA\_CON** query. One possible reason could be the third mentioned `SyntaxError`, where LeetCode considers function bodies with explicit type annotations behind variables as invalid syntax, decreasing the number of tests passed by **TA\_CON** query by 3.2%.

To explain why Copilot predicts more accurate completions for LeetCode than CodeCheck, we hypothesize that LeetCode problems appeared more on Copilot's training data than CodeCheck problems. One piece of evidence supporting our hypothesis is the fact that the predicted arguments for LeetCode's usage contexts are identical to the example arguments on the LeetCode website. In Figure 5.2, Copilot predicted the usage context argu-

```

1  def sumRowOrCol(input1: List[List[int]], input2: int, input3: str) -> int:
2      if input3 == 'row':
3          return sum(input1[input2])
4      else:
5          return sum([row[input2] for row in input1])

1  s: int = sumRowOrCol([[1,2,3],[4,5,6],[7,8,9]], 1, "R")
2  def sumRowOrCol(input1: List[List[int]], input2: int, input3: str) -> int:
3      if input3 == "R":
4          return sum(input1[input2])
5      else:
6          return sum([input1[i][input2] for i in range(len(input1))])

```

Figure 5.1: Similar Copilot’s completions for TA and TA\_CON, only compared strings are different

```

1  s = destCity([["London", "New York"], ["New York", "Lima"], ["Lima", "Sao Paulo"]])

```

Figure 5.2: Copilot’s predicted arguments for usage context are identical to the example arguments on the LeetCode website

ments `[["London", "New York"], ["New York", "Lima"], ["Lima", "Sao Paulo"]]` for function `destCity`, which is identical to the first example argument provided by LeetCode. We observed this tendency over 37/50 LeetCode problems, shown in bold in Table A.2. Therefore, we tested these 37 problems again to see whether providing different usage context arguments can reduce the accuracy of Copilot’s predictions and reported the results on Table A.3. The results show that CON query has a substantial decrease in accuracy, from 99% to only 47.6% test passed ( $\chi^2(2) = 10478.76, \rho < 0.001$ ), while TA\_CON query maintains the same accuracy (96%) ( $\chi^2(2) = 15.19, \rho < 0.001$ ). The reason behind these results may be that CON queries only rely on information about usage context, and it was most influenced by argument changes, whereas TA\_CON queries still have information from type annotation, allowing them to maintain high accuracy.

Although the predictions for LeetCode problems passed over 90% test cases, we anticipate that Copilot has the potential to achieve even higher accuracy if we keep the existing parameter names rather than changing them to `input`, as this can change Copilot’s probabilistic determination of what is likely to come next and generate suggestions.

## 5.2 RQ2: Complexity

Similar to our expectation, CodeCheck’s complexity varies among four queries and tends to show a negative correlation with the amount of information. The order of queries from highest to lowest complexity is NONE, TA and CON, TA\_CON. However, LeetCode’s complexity shows no substantial variation and has the same complexity median values of 4 across four queries. One reason for this occurrence could be that LeetCode problems were overly represented in Copilot’s training data, making it more likely that Copilot will

automatically produce identical completions when provided with LeetCode function names, regardless of any extra information about type annotations or usage contexts.

A prior study by Dakhel et al. [11] comparing Copilot-generated code with human completions found that Copilot often utilizes built-in methods in its suggested function bodies. In our study, we further discovered that these built-in methods are more frequently called in TA\_CON queries.



# Threats to Validity

---

In this chapter, we address the remaining limitations and threats to validity.

## 6.1 Internal Validity

The fact that Copilot is a closed-source model makes it difficult to determine whether and how frequently our tested programming problems appear in Copilot training data. This potentially introduces bias, as Copilot requires only function names to complete the function body for problems that frequently appear in the training data, regardless of any extra information about type annotations or usage contexts. An example of this is when Copilot predicted identical usage context arguments to LeetCode example arguments, even when only given the function name, as described in Section 5.1.

The procedure by which CodeCheck and LeetCode evaluate accuracy is different. While CodeCheck executes all test cases, LeetCode terminates when it encounters a fail test or an error and reports the lower bound. However, this has a minuscule impact on our study's results. Even though LeetCode reports lower bounds in some cases, it still has much higher accuracy than CodeCheck.

A notable threat to validity is the non-deterministic nature of LLMs like Copilot, raising concerns about the replicability of our results. Copilot is very sensitive to its input prompts. Anecdotally, we observed during our experiment different function body suggestions based on whether we included a colon at the end of the input code. To reduce this threat, we use a consistent prompt that is always including the colon.

According to the Copilot production team, Copilot makes predictions not only based on the code in the currently open file and the surrounding lines of the cursor, but also based on the information included in other files open in the editor and the URLs of repositories or file paths to identify relevant contexts [15]. As a result, it may be able to learn from the predictions triggered by other queries saved in other files.

Providing type annotations also provides information about the number of parameters and return variables, which means that TA, TA\_CON queries with type annotations always provide more information than NONE, CON queries with no type annotations. Even though we attempted to alleviate the imbalance by providing the number of parameters as input1, input2 as in Figure 4.3, we were unable to change the query from `def findTheDifference(input1, input2):` into `def findTheDifference(input1, input2) -> output:` since this is invalid Python syntax.

## 6.2 External Validity

Since we have to manually build query contexts for each programming problem and to fit into the scope of a bachelor thesis, our study only examined common Python data types and is limited to 100 programming problems, totaling 400 queries. Our results may not be generalizable due to these constraints.

# Concluding Remarks

---

## 7.1 Conclusion

LLMs can manage huge databases that humans cannot, unlocking multiple previously unknown potentials. Copilot is one such model, capable of synthesizing and autonomously generating context-related code from its massive GitHub training codebases. With Copilot now available as a commercial product, a new wave of developers will gain access to it. It is crucial for developers to understand how to modify their query context effectively to obtain higher-quality code suggestions and optimize Copilot as their ideal "AI pair programmer". Our study is the first to explore the impact of type annotations and usage context on Copilot's code completions. Overall, we found that Copilot's completions for CodeCheck problems have average accuracy and low complexity, with substantial variations between four query contexts. The amount of information in CodeCheck queries shows a positive correlation with Copilot's prediction accuracy and tends to be negatively correlated with Copilot's prediction complexity. **NONE** query has the lowest accuracy (41.4%) and the highest complexity (cyclomatic 3, cognitive 4), while **TA\_CON** query has the highest accuracy (72.8%) and the lowest complexity (cyclomatic 2, cognitive 1.5). LeetCode problems, on the other hand, have extremely high accuracy and low complexity, with no obvious variations between the four query contexts. The number of passed test cases for all four queries is above 90%, and their complexity shares a median of 4.

## 7.2 Future Work

We see various future work opportunities in looking into different kinds of information that could impact Copilot's predictions, other than type annotation and usage context. To fit into the scope of a Bachelor thesis, our study is limited to 100 Python programming problems from CodeCheck and LeetCode. Consequently, the results might not be generalizable and might exhibit limited statistical significance. Future study with a larger sample size is merited, as it reduces variability and may increase the test statistic, which in turn may make it easier to reach a threshold for significance. Additionally, further investigation into rarely encountered Python data types, such as dict, set, bytes, is justified to explore whether Copilot can maintain its accuracy when predicting across a broader range of data types. Copilot can not only suggest code completion but also generate comments and documents, as well as translate code between programming languages. These features are all highly useful for developers and should be further investigated.





## Appendix

---

Table A.1: Accuracy of Copilot’s completions for 50 CodeCheck problems. The table contains the number of tests each Copilot’s completion passed and its submission status. The bottom section contains the total tests passed and the submission status of 50 problems.

Problem	#Tests	Number (%) of test cases passed			
		No type annotation, no usage context	Type annotation, no usage context	No type annotation, usage context	Type annotation, usage context
Branches	valuesHaveTheSameSign	5 (100%)	4 (80%)	4 (80%)	4 (80%)
	coordinatesInSameQuadrant	6 (75%)	6 (75%)	6 (75%)	6 (75%)
	strict	3 (60%)	1 (20%)	0 (0%)	4 (80%)
	exactlyTwoTheSame	6 (100%)	6 (100%)	2 (33.3%)	2 (33.3%)
	closerToTarget	1 (16.7%)	5 (83.3%)	5 (83.3%)	5 (83.3%)
	areSameColor	3 (60%)	3 (60%)	0 (0%)	5 (100%)
	evenlySpaced	6 (100%)	6 (100%)	6 (100%)	6 (100%)
Strings	swapFirstAndLast	5 (100%)	3 (60%)	3 (60%)	3 (60%)
	reverseHalves	4 (57.1%)	4 (57.1%)	4 (57.1%)	4 (57.1%)
	countFrontBackMatches	2 (33.3%)	6 (100%)	6 (100%)	6 (100%)
	firstPositionDiffer	5 (100%)	3 (60%)	4 (80%)	4 (80%)
	removeMatchingPrefixSuffix	4 (57.1%)	2 (28.6%)	0 (0%)	4 (57.1%)
	numbersInside	0 (0%)	2 (66.7%)	2 (66.7%)	2 (66.7%)
	removeAdjacentDuplicates	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	firstDoubledVowel	2 (33.3%)	0 (0%)	2 (33.3%)	1 (16.7%)
Lists	evenThenOdd	0 (0%)	0 (0%)	0 (0%)	5 (100%)
	averageFirstTwoAndLastTwo	4 (100%)	4 (100%)	4 (100%)	4 (100%)
	repeatNumTimes	0 (0%)	2 (50%)	0 (0%)	2 (50%)
	swapMinAndMax	3 (100%)	3 (100%)	3 (100%)	3 (100%)
	positionOfLastLargest	4 (100%)	4 (100%)	4 (100%)	4 (100%)
	firstPositionSame	2 (40%)	5 (100%)	5 (100%)	5 (100%)
	findClosestValueIndex	1 (20%)	1 (20%)	1 (20%)	1 (20%)
	countInRange	0 (0%)	6 (100%)	6 (100%)	6 (100%)
	mostFrequentElement	5 (100%)	5 (100%)	5 (100%)	5 (100%)
	removeDuplicates	0 (0%)	6 (100%)	6 (100%)	6 (100%)
	firstRepeatedIndex	3 (50%)	3 (50%)	6 (100%)	6 (100%)
	averageOfConsecutivePairs	3 (100%)	3 (100%)	3 (100%)	0 (0%)
	averageWithoutMaxAndMin	2 (100%)	2 (100%)	2 (100%)	2 (100%)
	largestSumNConsecutive	2 (100%)	2 (100%)	2 (100%)	2 (100%)
	moveZerosToBack	6 (100%)	6 (100%)	5 (83.3%)	6 (100%)
	removeEvenElementsOccurringTwice	0 (0%)	1 (20%)	4 (80%)	4 (80%)
	removeElementsInRange	0 (0%)	0 (0%)	0 (0%)	1 (25%)
	splitEvenOddIndices	0 (0%)	5 (100%)	0 (0%)	5 (100%)
	getMinAndMax	0 (0%)	5 (100%)	0 (0%)	5 (100%)
	largestReverseExists	0 (0%)	2 (40%)	0 (0%)	2 (40%)
	largestConsecutiveSum	4 (100%)	4 (100%)	4 (100%)	4 (100%)
Two-Dimensional Arrays	averageOfCorners	5 (100%)	5 (100%)	5 (100%)	5 (100%)
	getChessBoardInfo	0 (0%)	0 (0%)	2 (40%)	2 (40%)
	sumRowOrCol	0 (0%)	3 (60%)	5 (100%)	5 (100%)
	sumDiagonals	2 (50%)	4 (100%)	2 (50%)	2 (50%)
	sumBordersWithoutCorners	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	countNegativeByRow	0 (0%)	4 (100%)	0 (0%)	4 (100%)
	findLocation	0 (0%)	5 (100%)	0 (0%)	5 (100%)
	slideRight	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	replaceNegativesWithZero	0 (0%)	5 (100%)	5 (100%)	5 (100%)
	largestNeighbor	0 (0%)	6 (100%)	6 (100%)	2 (33.3%)
	fillNeighbors	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	copyNeighbors	1 (33.3%)	1 (33.3%)	0 (0%)	1 (33.3%)
	getGreaterElements	0 (0%)	5 (100%)	5 (100%)	5 (100%)
	countDistinctElements	0 (0%)	4 (100%)	4 (100%)	4 (100%)
Total test cases passed		239 (41.4%)	162 (67.8%)	138 (57.7%)	174 (72.8%)
Accepted		15 (30%)	25 (50%)	20 (40%)	25 (50%)
Wrong answer		23 (46%)	21 (42%)	28 (56%)	22 (44%)
Runtime errors		12 (24%)	4 (8%)	2 (4%)	3 (6%)

Table A.2: Accuracy of Copilot’s completions for 50 LeetCode problems. The table contains the number of tests each Copilot’s completion passed and its submission status. Problems predicted with similar usage context arguments to the LeetCode website are in bold. The bottom section contains the total tests passed and the submission status of 50 problems.

Problem	#Tests	Number (%) of test cases passed			
		No type annotation, no usage context	Type annotation, no usage context	No type annotation, usage context	Type annotation, usage context
Easy	hammingDistance	149	0 (0%)	149 (100%)	149 (100%)
	constructRectangle	52	52 (100%)	52 (100%)	52 (100%)
	search	47	47 (100%)	47 (100%)	47 (100%)
	removeOuterParentheses	59	59 (100%)	59 (100%)	59 (100%)
	findTheDifference	54	0 (0%)	54 (100%)	0 (0%)
	isPerfectSquare	71	71 (100%)	71 (100%)	71 (100%)
	twoSum	63	63 (100%)	63 (100%)	63 (100%)
	removeElement	115	115 (100%)	115 (100%)	115 (100%)
	searchInsert	65	65 (100%)	65 (100%)	65 (100%)
	hasPathSum	117	117 (100%)	117 (100%)	117 (100%)
	isPalindrome	485	0 (0%)	485 (100%)	0 (0%)
	containsDuplicate	75	75 (100%)	75 (100%)	75 (100%)
	addDigits	1101	1101 (100%)	1101 (100%)	1101 (100%)
	countBits	15	0 (0%)	15 (100%)	0 (0%)
	fib	31	31 (100%)	31 (100%)	31 (100%)
	destCity	104	104 (100%)	104 (100%)	104 (100%)
	longestCommonPrefix	126	126 (100%)	126 (100%)	126 (100%)
Medium	threeSum	313	313 (100%)	313 (100%)	313 (100%)
	maxArea	62	62 (100%)	62 (100%)	62 (100%)
	threeSumClosest	102	102 (100%)	102 (100%)	102 (100%)
	isValidSudoku	507	507 (100%)	507 (100%)	507 (100%)
	nextPermutation	266	262 (98%)	266 (100%)	262 (98%)
	generateParenthesis	8	0 (0%)	8 (100%)	8 (100%)
	permute	26	0 (0%)	0 (0%)	0 (0%)
	combinationSum	160	160 (100%)	160 (100%)	160 (100%)
	myPow	306	306 (100%)	306 (100%)	306 (100%)
	rotate	21	1 (4.8%)	1 (4.8%)	1 (4.8%)
	removeNthFromEnd	208	208 (100%)	208 (100%)	208 (100%)
	swapPairs	55	0 (0%)	55 (100%)	55 (100%)
	uniquePaths	63	63 (100%)	63 (100%)	63 (100%)
	insert	157	0 (0%)	0 (0%)	0 (0%)
	merge	170	6 (3.5%)	6 (3.5%)	170 (100%)
	numDecodings	269	269 (100%)	269 (100%)	269 (100%)
	exist	87	0 (0%)	0 (0%)	0 (0%)
	minDistance	1146	1146 (100%)	1146 (100%)	1146 (100%)
	maximumGap	41	41 (100%)	41 (100%)	41 (100%)
	sortColors	87	87 (100%)	87 (100%)	87 (100%)
	rangeBitwiseAnd	8270	8270 (100%)	8270 (100%)	8270 (100%)
Hard	solveSudoku	6	0 (0%)	0 (0%)	0 (0%)
	largestRectangleArea	99	99 (100%)	99 (100%)	99 (100%)
	numDistinct	65	65 (100%)	65 (100%)	65 (100%)
	firstMissingPositive	177	177 (100%)	177 (100%)	177 (100%)
	bestRotation	44	44 (100%)	44 (100%)	44 (100%)
	containsNearbyAlmostDuplicate	52	30 (57.7%)	30 (57.7%)	30 (57.7%)
	smallestDistancePair	19	19 (100%)	19 (100%)	19 (100%)
	isNumber	1494	1487 (99.5%)	1487 (99.5%)	1487 (99.5%)
	isRationalEqual	74	0 (0%)	0 (0%)	0 (0%)
	dieSimulator	32	0 (0%)	2 (6.3%)	1 (3.1%)
	shortestPath	55	0 (0%)	5 (9.1%)	1 (1.8%)
	maxSum	82	0 (0%)	0 (0%)	22 (26.8%)
	maxSum	82	0 (0%)	0 (0%)	22 (26.8%)
Total test cases passed		17252	15750 (91.3%)	16473 (95.5%)	16204 (93.9%)
Accepted			31 (62%)	37 (74%)	36 (72%)
Wrong answer			7 (14%)	8 (16%)	9 (18%)
Time limit exceeded			0 (0%)	0 (0%)	0 (0%)
Memory limit exceeded			1 (2%)	0 (0%)	0 (0%)
Compile errors			0 (0%)	0 (0%)	0 (0%)
Runtime errors			11 (22%)	5 (10%)	9 (18%)

Table A.3: Accuracy of Copilot’s completions for 37 LeetCode problems, for which Copilot predicted similar usage context arguments to the LeetCode website. The table contains the number of tests each Copilot’s completion passed, and its submission status. The bottom section contains the total tests passed and the submission status of 37 problems.

Problem	#Tests	Number (%) of test cases passed			
		No type annotation, usage context	No type annotation, different usage context	Type annotation, usage context	Type annotation, different usage context
Easy	hammingDistance	149 (100%)	149 (100%)	149 (100%)	149 (100%)
	constructRectangle	52 (100%)	52 (100%)	52 (100%)	52 (100%)
	removeOuterParentheses	59 (100%)	59 (100%)	59 (100%)	59 (100%)
	findTheDifference	54 (100%)	54 (100%)	0 (0%)	2 (3.7%)
	isPerfectSquare	71 (100%)	8 (11.3%)	71 (100%)	9 (12.7%)
	twoSum	63 (100%)	63 (100%)	63 (100%)	63 (100%)
	removeElement	115 (100%)	86 (74.8%)	115 (100%)	115 (100%)
	searchInsert	65 (100%)	65 (100%)	65 (100%)	65 (100%)
	hasPathSum	117 (100%)	117 (100%)	117 (100%)	117 (100%)
	containsDuplicate	75 (100%)	75 (100%)	75 (100%)	75 (100%)
	addDigits	1101 (100%)	1101 (100%)	1101 (100%)	1101 (100%)
	countBits	15 (100%)	0 (0%)	0 (0%)	15 (100%)
	destCity	104 (100%)	0 (0%)	104 (100%)	104 (100%)
	longestCommonPrefix	126 (100%)	126 (100%)	126 (100%)	126 (100%)
Medium	threeSum	313 (100%)	313 (100%)	313 (100%)	313 (100%)
	maxArea	62 (100%)	52 (83.9%)	62 (100%)	62 (100%)
	threeSumClosest	102 (100%)	102 (100%)	102 (100%)	102 (100%)
	nextPermutation	266 (98.5%)	138 (51.9%)	0 (0%)	266 (100%)
	generateParenthesis	8 (100%)	8 (100%)	0 (0%)	0 (0%)
	combinationSum	160 (100%)	160 (100%)	0 (0%)	160 (100%)
	myPow	306 (100%)	306 (100%)	306 (100%)	306 (100%)
	removeNthFromEnd	208 (100%)	202 (97.1%)	208 (100%)	208 (100%)
	uniquePaths	63 (100%)	37 (58.7%)	63 (100%)	63 (100%)
	numDecodings	269 (100%)	22 (8.2%)	269 (100%)	269 (100%)
	minDistance	1146 (100%)	24 (2.1%)	1146 (100%)	1146 (100%)
	maximumGap	41 (100%)	41 (100%)	41 (100%)	41 (100%)
	sortColors	87 (100%)	87 (100%)	87 (100%)	0 (0%)
	rangeBitwiseAnd	8270 (100%)	2193 (26.5%)	8270 (100%)	8270 (100%)
Hard	largestRectangleArea	99 (100%)	99 (100%)	99 (100%)	99 (100%)
	numDistinct	65 (100%)	0 (0%)	65 (100%)	0 (0%)
	firstMissingPositive	177 (100%)	66 (37.3%)	177 (100%)	81 (45.8%)
	bestRotation	44 (100%)	44 (100%)	44 (100%)	44 (100%)
	containsNearbyAlmostDuplicate	52 (57.7%)	36 (69.2%)	36 (69.2%)	36 (69.2%)
	smallestDistancePair	19 (100%)	19 (100%)	19 (100%)	19 (100%)
	isNumber	1487 (99.5%)	1487 (99.5%)	1494 (100%)	1491 (99.8%)
	isRationalEqual	74 (0%)	0 (0%)	0 (0%)	0 (0%)
	dieSimulator	32 (3.1%)	0 (0%)	1 (3.1%)	0 (0%)
Total test cases passed		15523 (99%)	7391 (47.6%)	14899 (96%)	15028 (96.8%)
Accepted		31 (83.8%)	20 (54.1%)	29 (78.4%)	27 (73%)
Wrong answer		4 (10.8%)	12 (32.4%)	2 (5.4%)	6 (16.2%)
Time limit exceeded		0 (0%)	3 (8.1%)	0 (0%)	0 (0%)
Memory limit exceeded		0 (0%)	0 (0%)	0 (0%)	1 (2.7%)
Compile errors		0 (0%)	0 (0%)	0 (0%)	0 (0%)
Runtime errors		2 (5.4%)	2 (5.4%)	6 (16.2%)	3 (8.1%)

# Bibliography

---

- [1] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. “Typilus: Neural Type Hints.” In: *Proceedings of the 41st ACM Sigplan Conference on Programming Language Design and Implementation*. 2020, pp. 91–105.
- [2] Jong-hoon An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. “Dynamic Inference of Static Types for Ruby.” In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 459–472.
- [3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. “Towards Type Inference for JavaScript.” In: *ECOOP 2005-Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings* 19. Springer. 2005, pp. 428–452.
- [4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. “Language Models Are Few-Shot Learners.” In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901.
- [5] G Ann Campbell. “Cognitive Complexity-A New Way of Measuring Understandability.” In: *SonarSource SA* (2018), p. 10.
- [6] G Ann Campbell. “Cognitive Complexity: An Overview and Evaluation.” In: *Proceedings of the 2018 International Conference on Technical Debt*. 2018, pp. 57–58.
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. “Evaluating Large Language Models Trained on Code.” In: *arXiv preprint arXiv:2107.03374* (2021).
- [8] Shi-Yi Chen, Zhe Feng, and Xiaolian Yi. “A General Introduction to Adjustment for Multiple Comparisons.” In: *Journal of Thoracic Disease* 9.6 (2017), p. 1725.
- [9] *CodeCheck Python Exercises*. [Online; accessed 13-August-2024]. URL: <https://horstmann.com/codecheck/python-questions.html>.
- [10] *CodeCheck*. [Online; accessed 13-August-2024]. URL: <https://horstmann.com/codecheck/>.
- [11] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. “GitHub Copilot AI Pair Programmer: Asset or Liability?” In: *Journal of Systems and Software* 203 (2023), p. 111734.
- [12] Paul Denny, Viraj Kumar, and Nasser Giacaman. “Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language.” In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 1136–1142.

- [13] Luca Di Grazia and Michael Pradel. “The Evolution of Type Annotations in Python: An Empirical Study.” In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 209–220.
- [14] Michael Furr, Jong-hoon An, and Jeffrey S Foster. “Profile-Guided Static Typing for Dynamic Scripting Languages.” In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. 2009, pp. 283–300.
- [15] *GitHub Copilot - Your AI pair programmer*. [Online; accessed 13-August-2024]. 2021. URL: <https://copilot.github.com/>.
- [16] *GitHub Education*. [Online; accessed 13-August-2024]. URL: <https://github.com/education/>.
- [17] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. “An Empirical Study on the Impact of Static Typing on Software Maintainability.” In: *Empirical Software Engineering* 19.5 (2014), pp. 1335–1382.
- [18] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. “Deep Learning Type Inference.” In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 152–162.
- [19] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. “On the Naturalness of Software.” In: *Communications of the ACM* 59.5 (2016), pp. 122–131.
- [20] Jason Hsu. *Multiple comparisons: theory and methods*. CRC Press, 1996.
- [21] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “Type Analysis for JavaScript.” In: *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings* 16. Springer. 2009, pp. 238–255.
- [22] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. “Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code.” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1073–1085.
- [23] Deepak Kumar. “REFLECTIONS Tools from the Education Industry.” In: *ACM inroads* 9.3 (2018), pp. 22–24.
- [24] *LeetCode Problems*. [Online; accessed 13-August-2024]. URL: <https://leetcode.com/problemset/>.
- [25] *LeetCode*. [Online; accessed 13-August-2024]. URL: <https://leetcode.com/>.
- [26] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. “NL2Type: Inferring JavaScript Function Types From Natural Language Information.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 304–315.
- [27] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. “Portfolio: Finding Relevant Functions and Their Usage.” In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 111–120.
- [28] Ekaterina A Moroz, Vladimir O Grizkevich, and Igor M Novozhilov. “The Potential of Artificial Intelligence as a Method of Software Developer’s Productivity Improvement.” In: *2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. IEEE. 2022, pp. 386–390.

- [29] Nhan Nguyen and Sarah Nadi. "An Empirical Evaluation of GitHub Copilot's Code Suggestions." In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 1–5.
- [30] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. "Typewriter: Neural Type Prediction With Search-Based Validation." In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 209–220.
- [31] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. "Zero-Shot Text-to-Image Generation." In: *International Conference on Machine Learning*. Pmlr. 2021, pp. 8821–8831.
- [32] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. "A Large Scale Study of Programming Languages and Code Quality in GitHub." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 155–165.
- [33] Veselin Raychev, Pavol Bielik, and Martin Vechev. "Probabilistic Model for Code With Decision Trees." In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 731–747.
- [34] Veselin Raychev, Martin Vechev, and Andreas Krause. "Predicting program properties from" big code"." In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 111–124.
- [35] Brianna M Ren, John Toman, T Stephen Strickland, and Jeffrey S Foster. "The Ruby Type Checker." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013, pp. 1565–1572.
- [36] SORA. [Online; accessed 13-August-2024]. URL: <https://openai.com/index/sora/>.
- [37] Jonathan Sillito, Gail C Murphy, and Kris De Volder. "Questions Programmers Ask During Software Evolution Tasks." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2006, pp. 23–34.
- [38] Jonathan Sillito, Gail C Murphy, and Kris De Volder. "Asking and Answering Questions During a Programming Change Task." In: *IEEE Transactions on Software Engineering* 34.4 (2008), pp. 434–451.
- [39] Susan Elliott Sim, Charles LA Clarke, and Richard C Holt. "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers." In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE. 1998, pp. 180–187.
- [40] SonarQube Metric Definitions. [Online; accessed 13-August-2024]. URL: <https://docs.sonarsource.com/sonarqube/latest/user-guide/code-metrics/metrics-definition/#complexity>.
- [41] SonarQube. [Online; accessed 13-August-2024]. URL: <https://www.sonarsource.com/products/sonarqube/>.
- [42] TIOBE Index. [Online; accessed 13-August-2024]. URL: <https://www.tiobe.com/tiobe-index/>.
- [43] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. "On the Localness of Software." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 269–280.

- [44] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. “Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models.” In: *Chi Conference on Human Factors in Computing Systems Extended Abstracts*. 2022, pp. 1–7.
- [45] Michel Wermelinger. “Using GitHub Copilot To Solve Simple Programming Problems.” In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 172–178.
- [46] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. “Python Probabilistic Type Inference With Natural Language Support.” In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 607–618.