Evaluating Copilot Lab's Code Explanation Capabilities

Benxuan Zhao

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

August 23, 2023

Abstract

In the current programming environment, code explanation and documentation are critical to developer and end-user understanding. This study evaluates the quality of code explanations through a unique approach: regenerating code by utilizing Copilot Lab-generated code explanations and comparing this code to the original code. We used the Copilot plugin installed in IDEA as the main code generation tool to ensure the impartiality of the experiment. By evaluating the accuracy, similarity, and performance of the re-generated code, we indirectly assess the accuracy and validity of the code explanation. Preliminary results suggest that methods measured through proxies can provide valuable insights for assessing the quality of code explanations. This provides an important reference for the development of future code generation and documentation tools.

Education Use Consent

Acknowledgements

Many thanks to my supervisor Handan Gul Calikli for supervising and guiding my dissertation during these months. She was the one who initially provided me with ideas for my dissertation topic, and then she continued to encourage me and give me advice, and was quite accommodating to the mistakes I made.

I would also like to thank my family and friends, Lingyun Li and Hao Wang, for their help and encouragement even though they are far away from China, so that I can concentrate more on completing my thesis.

Contents

Chapter 1 Introduction

Artificial intelligence has been in the spotlight in recent years simply because of the remarkable advances it has made in a variety of fields, none more so than ChatGPT, whose creator team, OpenAI, has long partnered with github to launch Copilot, which focuses on software development.Copilot is designed to assist developers in writing code by generating suggestions and complete code snippets based on contexts and patterns. complete code snippets to assist developers in writing code. With its ability to understand programming languages and provide intelligent code suggestions, Copilot has become a valuable asset to many developers.

To further enhance Copilot, OpenAI has launched Copilot Lab, which has a new experimental feature, code explanation. Programmers know that understanding of existing code is crucial for software development, especially for novice developers and new colleagues. This feature can help them better understand the logic of existing code, which can help them develop the right code or documentation.

Copilot Lab's Code Explanation feature received a great deal of attention on the web as soon as it went live, generating a wide range of discussions among programmers in the developer community and on self-publishing platforms, anticipating its potential and doubting its misleading and limiting nature.

In this study, considering the interest and the importance of code understanding, a comprehensive analysis of the usability of Copilot Lab's code explanation function will be carried out, including the generation of explanations for different types of code, the natural language analysis of the code explanations for correctness and comprehensibility, and the quality and variance analysis of the re-generation of the code based on the code explanations with the aim of assessing whether it provides a high-quality reference for software development.

The remaining chapters of this dissertation detail the project as follows.

- Chapter 2 describes the tools and experimental methods used by copilot and its team of creators to analyze the contribution of the code explanation feature of copilot lab to the understanding of the code.

- Chapter 3 describes the specific aspects of the experiments and evaluations carried out to analyze the usability of copilot lab's code explanation features, how these experiments and evaluations were implemented, and the reasons and necessity for the existence of complementary relationships between these experiments and evaluations.

- Chapter 4 shows the code explanations generated by copilot lab as natural language and the data obtained from the experiments, and generalizes and analyzes this natural language and data.

- Chapter 5 summarizes the experimental process and analysis of the copilot lab code explanation feature conducted in this study and discusses the possible future development of this feature and its impact on the software development industry.

Chapter 2 Background

This chapter describes the role of understanding code in the field of software development, the context in which the COPILOT LAB was created, how it differs from COPILOT, the software analysis tools used, and an overview of related papers.

2.1 Understanding What The Code Does

2.1.1 Diagnosis And Debugging Of Problems

Understanding the code you're writing and the frameworks or external libraries you're using can make your code clearer, and make it easier to pinpoint the location of the error when a problem arises, which makes debugging code more efficient.

2.1.2 Maintenance And Iteration

For active system code that needs to be maintained, it's important to understand how it works and what it was designed to do, so that you can fix any bugs that arise without introducing new ones. If you need to iterate on the original system to add new functionality, having a good understanding will also prevent you from introducing redundant code.

2.1.3 Learning And Enhancement

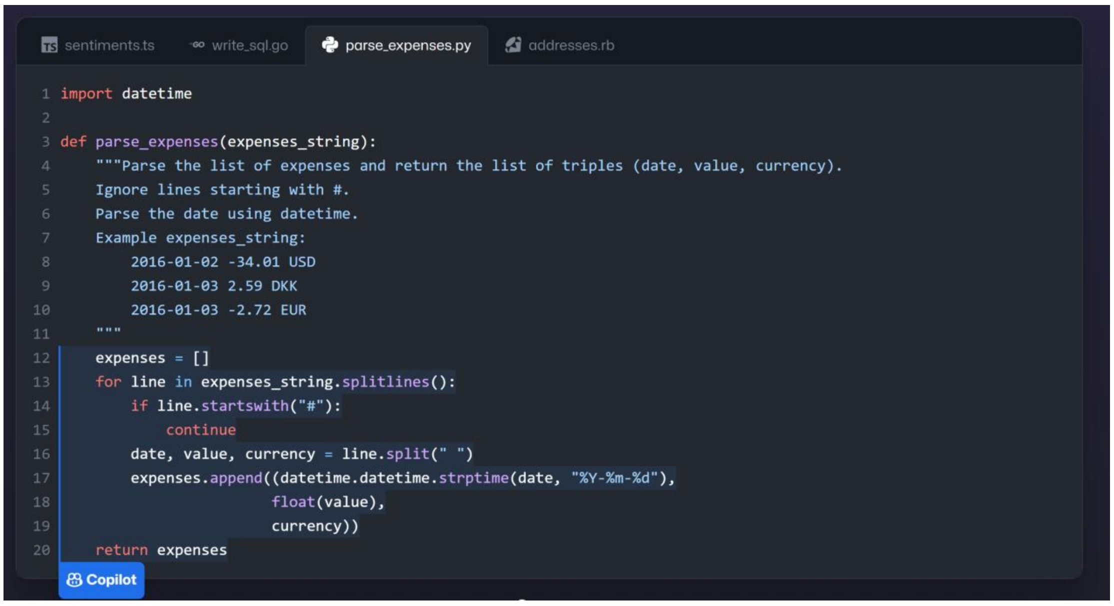
Understanding code written by others is undoubtedly the key to improving one's own skills, which can only be achieved by learning and imitating high-level code, including design patterns, algorithms and data structures, and even formatting and comments.

2.1.4 Code Quality

Only by understanding the code in front of you can you add comments or documentation to it. It is this fully readable and maintainable system that will prevent it from becoming a legacy system years down the road.

2.2 GitHub Copilot And Copilot Lab

Copilot is an AI-assisted programming tool developed by OpenAI and GitHub, based on a large amount of open source code training, with the aim of providing suggestions and parts of the code while programming, and even learning the user's code style to improve the developer's programming efficiency.



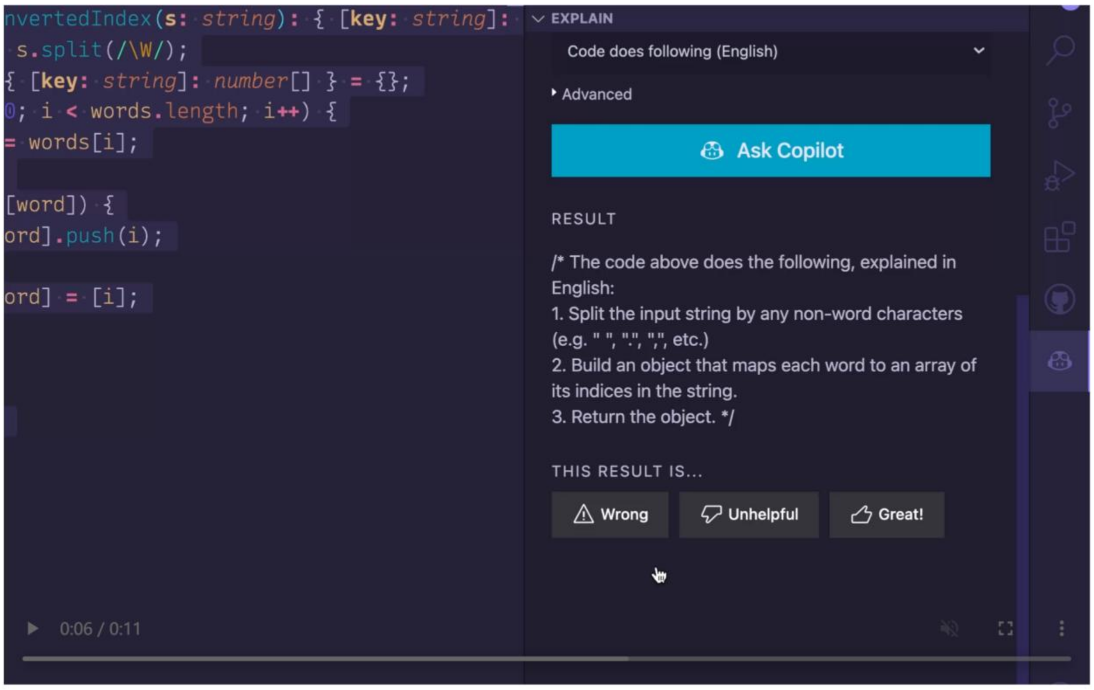
```python
import datetime

def parse_expenses(expenses_string):
    """Parse the list of expenses and return the list of triples (date, value, currency).
    Ignore lines starting with #.
    Parse the date using datetime.
    Example expenses_string:
        2016-01-02 -34.01 USD
        2016-01-03 2.59 DKK
        2016-01-03 -2.72 EUR
    """
    expenses = []
    for line in expenses_string.splitlines():
        if line.startswith("#"):
            continue
        date, value, currency = line.split(" ")
        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
                         float(value),
                         currency))
    return expenses
```

Figure 1: Copilot Official Demonstration

As shown in Figure 1, Copilot can generate method content based on code comments and method definitions.

Copilot Lab is an extension plugin that relies on GitHub Copilot and stands on its own, it can be used in VS Code and is a testing ground for Copilot to provide more functionality for developers. Its fundamental purpose is also to improve the efficiency and quality of developers writing code.

Copilot Lab includes features such as interpreting code, translating code, and generating tests, and is currently in the stage of usable prototypes, as shown in Figure 2, which is an official demo of the interpreting code feature that will be the focus of this paper. It provides support for multiple languages, the ability to choose different ways of interpreting code, and the ability to customize prompt words.


```
nvertedIndex(s: string): { [key: string]:
 s.split(/\W/);
{ [key: string]: number[] } = {};
0; i < words.length; i++) {
= words[i];

[word]) {
ord].push(i);

ord] = [i];
```

/* The code above does the following, explained in English:
1. Split the input string by any non-word characters (e.g. " ", ".", ",", etc.)
2. Build an object that maps each word to an array of its indices in the string.
3. Return the object. */

Figure 2: Copilot Lab Official Demonstration

2.3 Sonarqube

Sonarqube is a commonly used code quality management system, is a member of the Sonar family, it supports more than 30 programming languages, used to automatically review the code and systematically give the code bugs vulnerabilities duplicated and out of specification report by the testers feedback to the developers to improve the quality of the code and maintainability.

Sonarqube scans the project folder that needs to be statically analyzed by using SonarScanner to specify the project's compiled .class that is necessary for this project. It will automatically analyze the code (whether complete and runnable or not) for potential vulnerabilities and security issues.

2.4 ChatGPT

ChatGPT is an AI language model developed by OpenAI, trained on a large amount of textual data, including data, websites, and other public textual information.

ChatGPT can carry out conversations with text messages such as composing emails, code, quizzes, and even learning new text data given by the user. ChatGPT 4.0 was used in this study and utilized these capabilities to do the following three things:

1. supplemental generation of code that Copilot could not generate.

2. analyze the Copilot Lab generated annotations for text quality and comprehensibility.

3. give suggestions for modifying bugs in the generated code.。

2.5 Proxy Measurement

Proxy measurement technique is a method of indirect measurement of target parameters. Its basic idea is that when it is difficult or impossible to measure the target parameter directly, it is possible to choose a proxy variable that has a definite relationship with the target parameter to carry out the measurement, so as to obtain the value of the target parameter indirectly.

This technique is typically applied in the following situations.

The target parameter is difficult or impossible to measure directly. For example, in medical research, direct measurement of the functional status of certain organs may require invasive tests, in which case some blood indicators can be chosen as proxy variables.

The precision of measurement of target parameters is low. The use of proxy variables can improve the accuracy of measurement. For example, the magnitude of stars can be used as a proxy for stellar brightness.

The cost of target parameter measurement is high. Using proxy variables can reduce the measurement cost.

Target parameters cannot be measured repeatedly. It can be analyzed by repeated measurements of proxy variables.

In this study, since the explanations generated by Copilot Lab do not facilitate a direct assessment of quality, a proxy measure was used to assess the quality of the explanations by measuring the quality of the code that was re-generated using the explanations.

Chapter 3 Design

This study tests the quality and reliability of Copilot Lab's generation of explanations for Java code, the main processes are selecting code snippets, generating explanations, code re-generation using the explanations, evaluating the explanations and the code, collecting the data, and finally analyzing the collected data on the explanations and the code. As shown in Fig. 3 is the flowchart of this research design.

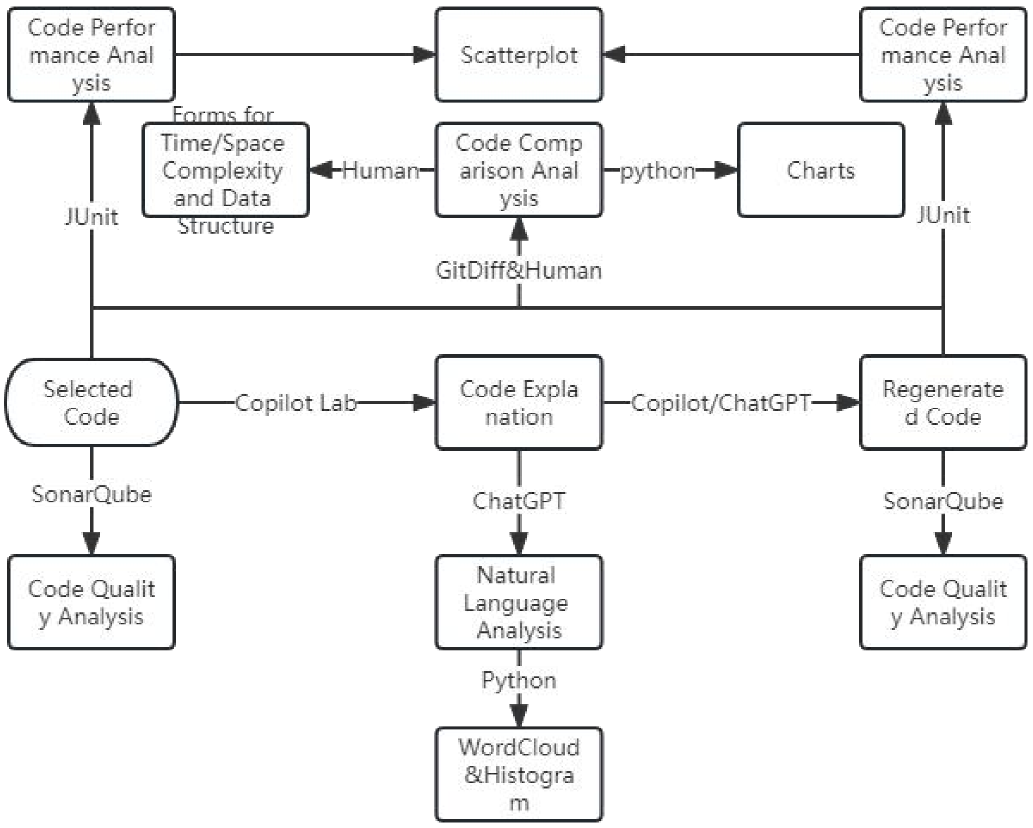


Figure 3: Experimental Flow Chart

3.1 Select Code Snippet

Two parts of the algorithmic code and the back-end project code, both from github, were chosen for this study, which gives them typicality and some complexity.

Algorithm code for leetcode part of the answer to the question, leetcode covers a lot of typical data structures and algorithms of the topic,, there is no lack of complex algorithms, as the data set is more representative. In order to make the subsequent statistical analysis more targeted, this study categorizes the algorithmic code into array operations, tree operations, double pointers, dynamic planning, fast and slow pointers, hashing, heaps, list operations, numerical operations, partitioning, recursion, retrospect, stack team, string operations, threads and others totaling 16 categories.

The back-end project code is a simple management system, covering database operation, business logic processing, api design and other aspects, and there are the original author's personal development habits, including code style and architectural design. This part of the code is typical and complex enough to represent Copilot Lab's performance in interpreting the actual project code.

And in order to create a more-than-difficult environment for Copilot Lab, this study purposely removed all of the original comments from the code, with the goal of exploring its pure understanding of the code's logic and reflecting a more realistic picture of the feature.

3.2 Generate Explanations

This study generates explanations in two steps, the first step is to find a suitable cue word through a small portion of the code, and the second step then uses this cue word to generate explanations for all the codes.

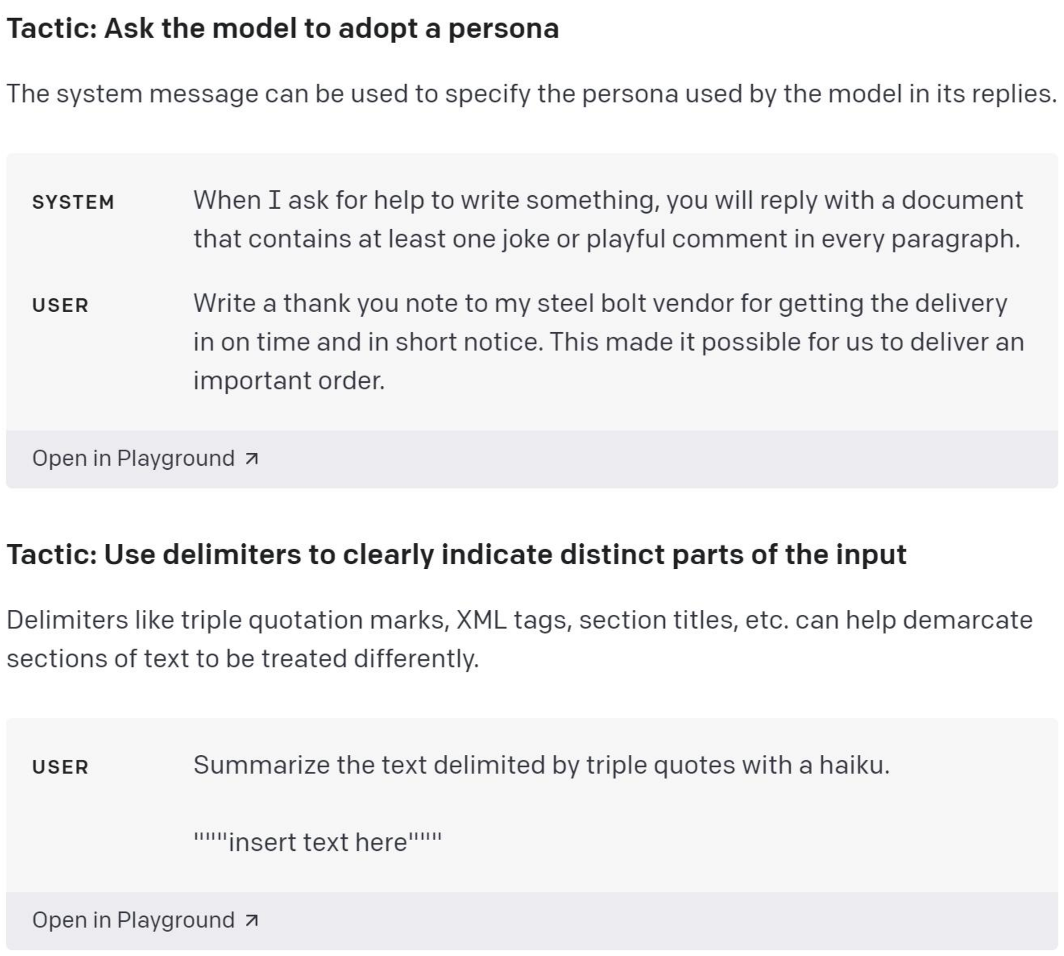
3.2.1 Debugging Prompts

Since Copilot Lab is context-sensitive and appropriate cue words are important for generating explanations, this experiment requires debugging prompts.

In this step, a portion of the simpler leetcode code is first used for debugging the prompt word. According to the methodology given on the official OpenAI website, there are six strategies: include detailed information in your query to get a more relevant answer, ask the model to adopt roles, use separators to clearly indicate the different parts of the input, specify the steps needed to complete the task, provide examples, and specify the desired length of the output.

By removing the non-compliant strategies according to the cue length limit, using the remaining strategies in conjunction with them to formulate cues, and iterating on these cues with modifications and default cues, it is expected to find a set of cues that can generate accurate and effective explanations through this process.

Figure 4 below shows an example of cue words given in the official OpenAI documentation.


Tactic: Ask the model to adopt a persona

The system message can be used to specify the persona used by the model in its replies.

SYSTEM When I ask for help to write something, you will reply with a document that contains at least one joke or playful comment in every paragraph.

USER Write a thank you note to my steel bolt vendor for getting the delivery in on time and in short notice. This made it possible for us to deliver an important order.

Tactic: Use delimiters to clearly indicate distinct parts of the input

Delimiters like triple quotation marks, XML tags, section titles, etc. can help demarcate sections of text to be treated differently.

USER Summarize the text delimited by triple quotes with a haiku.

"""insert text here"""

Figure 4: Example Of An Official OpenAI Prompt

3.2.2 Using Prompts To Generate Explanations

Once a set of hints has been identified, the next step is to generate an explanation of all the code using the hints.

This step involves entering the hints into Copilot Lab along with the corresponding code snippets, recording the generated code explanations, and also continuing to check the generated hints, each time an explanation is generated, to see if it meets expectations, effectively explains a key part of the code, or if there is an error, and if it is found, and it still doesn't work after several attempts, the code will be flagged and other hints will be used on it. use other hint words.

This part of the code and the generated explanations will be specifically analyzed in the subsequent evaluation section to see if there are certain patterns or specific types to get some shortcomings on the Copilot Lab code explanation function.

In addition, since web projects contain too much context and have some complexity, for the code explanation of web projects, this study will be given to Copilot Lab to interpret at different granularities, from large to small, to observe at what granularity Copilot works properly and at what granularity it works better.

3.3 Code Regeneration

After collecting all the generated code explanations, this study enters the code re-generation phase, which involves using Copilot to convert the generated code explanations, one by one, back into code, and evaluating whether the generated explanations are sufficiently detailed and accurate based on the differences and quality of the code.

This process will reflect Copilot's ability to reproduce the code based on the explanations we generate, and this ability is directly related to the quality of the explanations we input, if the explanations are not generated with high quality, Copilot will not be able to accurately generate the code, and there may even be a situation where the code generation is not related to the original code, and the subsequent experiments in this experiment will assess the quality of the generated codes and their differences with the original code, indirectly assessing Copilot's accuracy. The quality of the generated codes and their differences with the original codes will be evaluated in this experiment to indirectly assess the ability of Copilot Lab to generate explanations.

In order to prevent Copilot from obtaining context in the same project and generating code with different environments and settings to bring about differences, this study will conduct new code generation in a project different from the original code to ensure the accuracy, fairness, and consistency of the test, and the generated code will be used for subsequent evaluation and analysis.

Due to the limitations placed on the environment in which Copilot obtains context, this experiment needs to be prepared to deal with failure, therefore, if Copilot consistently fails to generate valid code based on the explanation and is unable to supplement or correct its code on its own, this experiment will choose to switch to ChatGPT as an alternative to use ChatGPT for supplemental code generation and correction.

The process of regenerating code using ChatGPT is similar to using Code Copilot, and should control the same environment and settings, and use the same code explanation as Copilot as input. Of course, even with ChatGPT 4.0, there may be cases where the generated code does not meet the requirements, so it is important to try a few more times and keep a record of what happens each time.。

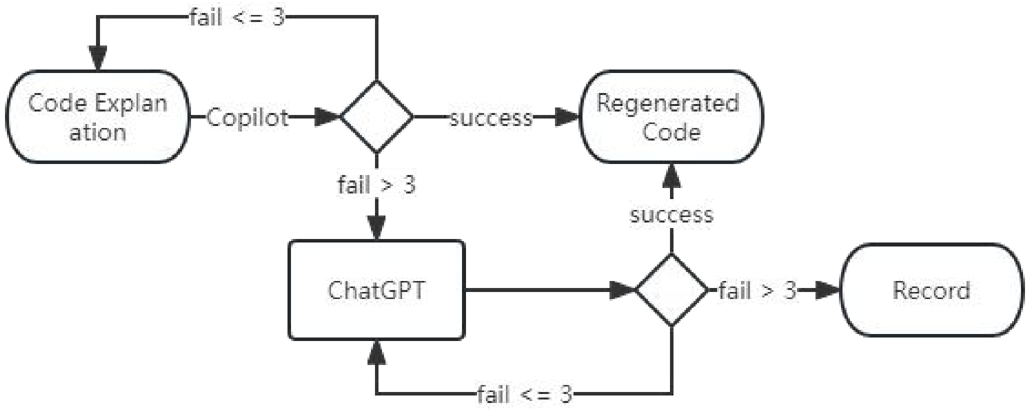


Figure 5: Re-generate the code flowchart

3.4 Evaluating Explanations And Codes

3.4.1 Evaluating Explanations

A direct assessment of explanations is the best indication of the level of Copilot Lab's code explanation function. Since it is difficult to find professional wordsmiths to evaluate the explanations, this study uses the textual language model ChatGPT 4.0 for the evaluation of the explanations.

In this process all the explanations were entered into ChatGPT one by one, and for each explanation, it was asked to say the role and its strengths and weaknesses in terms of natural language description.

Discrepancies between what ChatGPT says it does and what the code originally does, incompleteness, and errors (i.e., linguistic inaccuracies) are recorded, and, of course, the strengths and weaknesses of each explanation are recorded as well.

3.4.2 Evaluation Code

The evaluation of the code is divided into two parts, the first part is a direct evaluation of the quality of the re-generated code, which evaluates the re-generated code for the presence of bugs, irregularities, security hazards, and consistency of the logic with the description in the explanation.

The first three items are automatically evaluated in this study using the static code analysis tool SonarQube, while the consistency of the logic with that described in the explanation is manually checked.

The second part is a comparative assessment of the original and re-generated code, which is divided into the following three areas:

1. Comparison of logic, data structures and time-space complexity: first using GitDiff, but actually applying the integrated Git in the IDEA tool for code comparison, this step compares the logic, data structures used and time-space complexity of the two codes, and the expected effect of this step is to confirm that the generated code accurately replicates the original code's functionality and performance. A secondary validation of the differences in the implementation of the subjectively evaluated logic will also be performed using the Levenshtein distance to calculate the gap between the objective original code and the re-generated code.

2. SonarQube automated evaluation: SonarQube, a static code analysis tool, is then used to evaluate the quality of the original and generated code. The tool automatically detects bugs, vulnerabilities, bad flavors, and security hotspots in the code and gives a score. The purpose of this step is to provide a more in-depth , systematic evaluation , and further inspection of the generated code to meet high-quality software development standards.

3. JUnit Performance Evaluation: Finally, JUnit test cases are manually written to evaluate the performance of the original and re-generated code. This step will simulate the input and output of the runtime to test the robustness, stability and efficiency of the code. The desired effect is to determine whether the regenerated

code can give the same response as the original code when encountering various situations.

3.5 Data Collection

In this study, data collection is quite a critical step, as multiple IDEs, AIs, and software development tools were used, the output of the experiments was difficult to automate, so manual data collection was used, along with verification to confirm that each data point collected was accurate and complete. This data includes, but is not limited to, generated explanations, code snippets, evaluation results, and possible problems. Data from each step will be organized and saved in specific files or forms for subsequent analysis.

The specific steps are:

1. Data Generation: Based on the pre-designed process and inputs, the tool used will automatically generate data which will be recorded.

2. Data organization: After the experiment is over, all the generated data and the problems that occurred will be organized and saved in a document or a form. This process will clean inappropriate or duplicate data.

3. Data Import: After finishing the collation, all data will be copied into python lists or dictionaries for easy analysis calling and visualization.

3.6 Data Analysis Methods

Data analysis is very important in this study and it is the key to ensure the reliability of the study. There are two main methods of analysis used in this study, the first is using SonarQube which was mentioned earlier. The second is the visualization and analysis of the data using python.

There are three types of data analysis using python, descriptive analysis, data visualization and correlation analysis.

Descriptive analysis gets the basic characteristics and distribution of the data by calculating basic quantities such as central tendency and dispersion of the data.

Data visualization uses matplotlib to graphically display the data for a more intuitive understanding of the data.

Correlation analysis looks to find possible relationships between the time-space complexity of the code, the quality of the explanation and the quality of the re-generated code. However, due to the limited data obtained in this experiment and the inability to find enough participants in the experiment to score the natural language and re-generated code, this type of analysis only used ChatGPT to score the explanation, with the scores categorized as Grammar Accuracy, Coherence of Information, Consistency The scores are categorized into Grammar Accuracy, Coherence of Information, Consistency throughout the Text, Transitional Devices, and Logical Distribution of Information. The re-generated code only counts the difference with the original code, which is categorized as identical, logically identical, logically essentially identical, logically different and re-generation failure. In addition, regarding the time-space complexity, since there are too many types to be conveniently calculated, this experiment only considers the case of large-scale computation, and approximates the cases of O(MN + b) as O(N^2), O(MAX(N,M)) and O(M + N) as O(N). And the correlation analysis in this study is qualitative only。

Here are the role-play scoring commands used for ChatGPT.

Suppose you are an expert in natural language, below I have some code explanations generated by copilot lab based on the code, programmers need to follow this explanation to generate code, please rate these explanations 1-5

The correlation analysis was performed using Pearson's correlation coefficient analysis, which was designed to demonstrate that it is feasible to analyze the re-generated code in comparison to the original code in this study.

Chapter 4 Implementation

4.1 Explanation Generation

At the beginning of this step, it was first necessary to get access to Copilot and Copilot Lab, i.e., Github certified student status and a test qualification application for Copilot Lab, which took me a week to wait for the certification to go through. Only then could the formal implementation part begin.

4.1.1 Debugging Prompts

In this step, we tested the cue word strategy and the combination strategy in Table 1 several times and made at least five modifications for each strategy, concluding that using the default cue word is more conducive to generating code explanations with higher quality and success rates.

Table 1: Tactics And Performances.

Tactics	Performances
Include details	Frequently need to change prompt
Adopt a persona	Personification can get weird and a little verbose
Use delimiters	The effect is similar to the default
Provide examples	The effect is similar to the default
Include details+adopt a persona	Frequently need to change prompt
Include details+Use delimiters	Frequently need to change prompt
Include details+Provide examples	Frequently need to change prompt
Adopt a persona+Use delimiters	Personification can get weird
Adopt a persona+Provide examples	Personification can get weird
Use delimiters+Provide examples	Rarely better than default, but too detailed
Include details+Adopt a persona+Use	Frequently need to change prompt

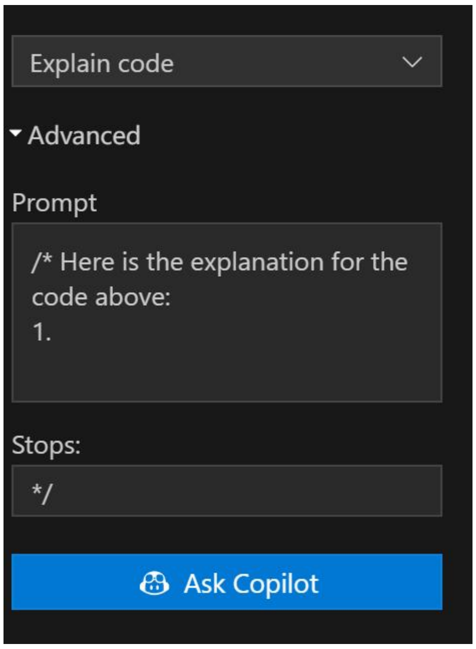
delimiters	
All of above	Often fails

This phenomenon may be due to the fact that the default cue word also uses the Use delimiters strategy and gives an ordinal number as a guide to the explanations that follow, and that the default cue word is widely used in the training process, so the model has some kind of optimization for it.

In summary, it was decided to use the default cue words to generate explanations in the follow-up.

4.1.2 Generate Explanations

First of all, from github to find the relevant code copied to the local and install the relevant dependencies, in this process you need to filter out duplicates and problematic code, remove the existing comments in the code, and categorized into different packages. In this experiment, more than 60 leetcode algorithm codes and a simple springboot framework web project are selected for subsequent operations. Open the project in VS Code, select the code that needs to generate an explanation, and use Copilot Lab to select the default prompt for explanation generation.



Explain code

Advanced

Prompt

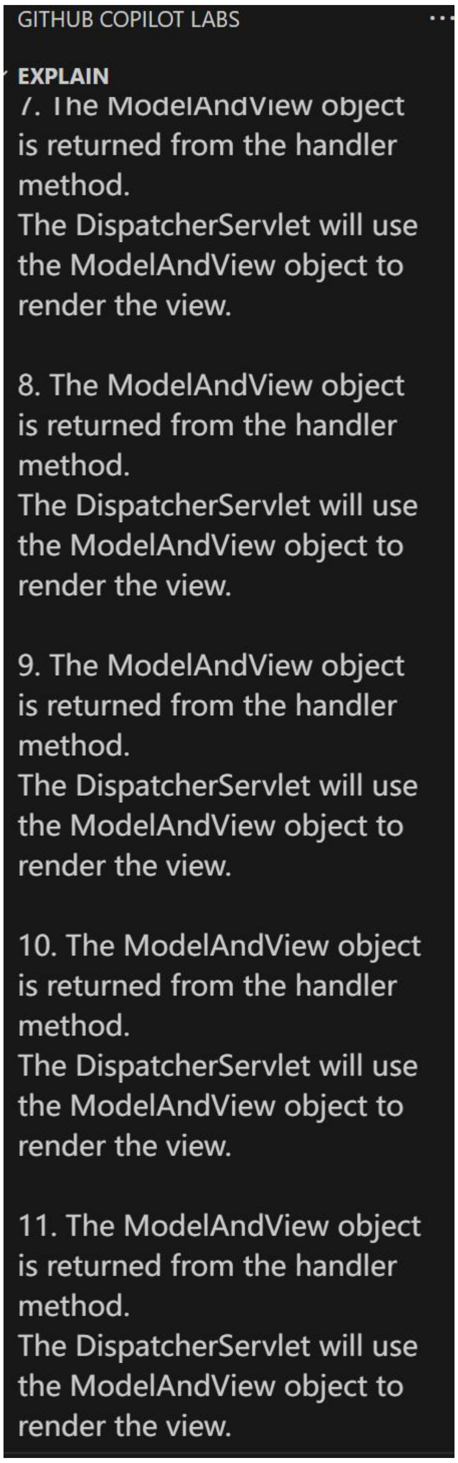
```
/* Here is the explanation for the code above:
1.
```

Stops:

```
*/
```

Figure 6: Copilot Lab Prompt

Of course, even with the best-tested prompt-generated explanations, errors sometimes occur, as shown in Figure 7, a problematic explanation of one of the interfaces of the web project's ItemController, which repeats one of the steps over and over again.


GITHUB COPILOT LABS

EXPLAIN

7. The ModelAndView object is returned from the handler method.
The DispatcherServlet will use the ModelAndView object to render the view.

8. The ModelAndView object is returned from the handler method.
The DispatcherServlet will use the ModelAndView object to render the view.

9. The ModelAndView object is returned from the handler method.
The DispatcherServlet will use the ModelAndView object to render the view.

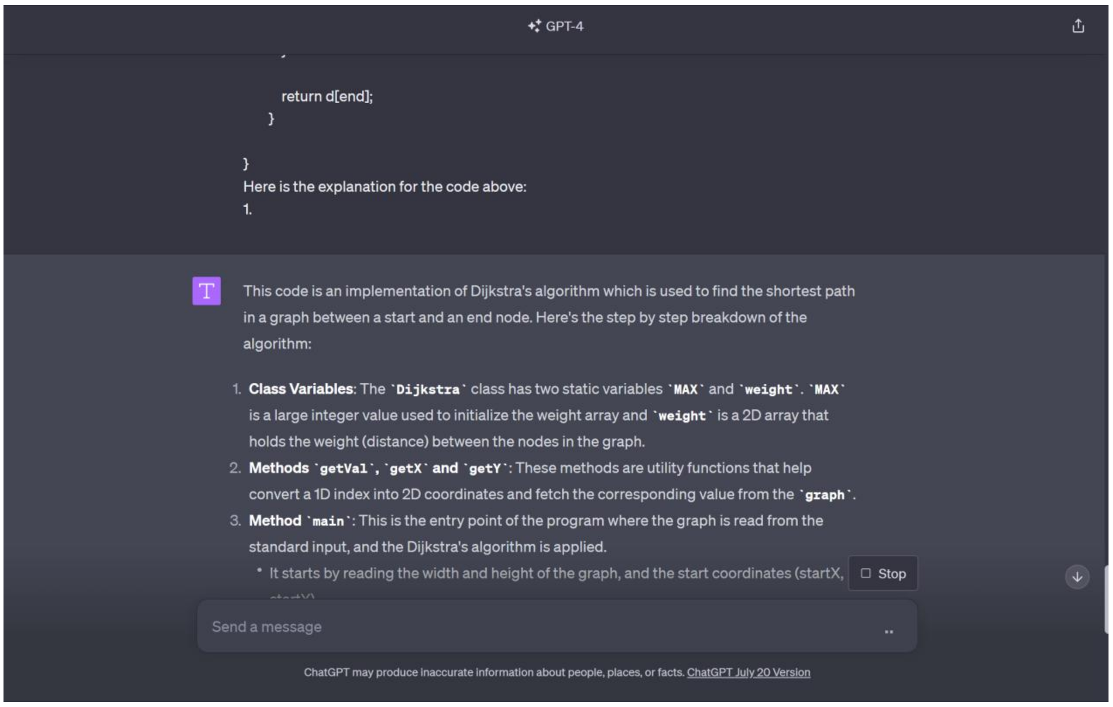
10. The ModelAndView object is returned from the handler method.
The DispatcherServlet will use the ModelAndView object to render the view.

11. The ModelAndView object is returned from the handler method.
The DispatcherServlet will use the ModelAndView object to render the view.

Figure 7: A Problem When Explain Code

Interpreting the algorithms project went relatively smoothly without too many errors, and if they did occur, they were resolved in retries, except for the Dijestra algorithm. The Dijestra algorithm generated a full Chinese explanation for the first explanation without any Chinese or Chinese abbreviations in the code, which is highlighted here. In the subsequent retries, the algorithm was also not successfully interpreted by Copilot Lab, and even appeared to report errors directly.

So this experiment will use the alternate plan to generate the explanation of this algorithm using ChatGPT, as shown in Figure 8 below, using the same prompt words as Copilot Lab. It is puzzling that ChatGPT showed significant lag, but this did not happen when the question was asked using Chinese. (Chinese was entered by mistake, but this seems to have some unspoken connection to the previous Copilot Lab-generated all-Chinese explanations.)



```
            return d[end];
        }

    }
```
Here is the explanation for the code above:
1.

This code is an implementation of Dijkstra's algorithm which is used to find the shortest path in a graph between a start and an end node. Here's the step by step breakdown of the algorithm:

1. **Class Variables**: The `Dijkstra` class has two static variables `MAX` and `weight`. `MAX` is a large integer value used to initialize the weight array and `weight` is a 2D array that holds the weight (distance) between the nodes in the graph.
2. **Methods `getVal`, `getX` and `getY`**: These methods are utility functions that help convert a 1D index into 2D coordinates and fetch the corresponding value from the `graph`.
3. **Method `main`**: This is the entry point of the program where the graph is read from the standard input, and the Dijkstra's algorithm is applied.
   * It starts by reading the width and height of the graph, and the start coordinates (startX,

ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT July 20 Version

Figure 8: ChatGPT Explains Dijkstra

This is an unavoidable mechanical repetitive process as Copilot Lab does not provide an API to support batch operations. All the generated explanations are documented along with the problems and errors that occurred during the process.

Also in this step, there is the code re-generation part of the web project, in which, due to the complexity of the project, this experiment takes a step-by-step approach to reduce the granularity until the correct code can be successfully generated at the class level and function level.

The first class-level granularity of the attempt to directly select all the code in a class, the use of Copilot Lab to generate the explanation, but unfortunately the success rate of this explanation is minimal, most of the cases will be reported directly to the error, the generated explanation is sometimes logically incorrect. However, for the sake of thoroughly exploring the limits of Copilot Lab's functionality, this experiment decided to keep this part of the successfully generated explanations and continue with the following steps.

So instead of using function-level granularity, the functions in a class were individually selected and Copilot Lab was used to generate explanations. Although there were still failures and irrationalities, they were only a few and could be corrected by re-running the code to explain the functionality.

After documenting all the explanations and issues that arose during the experiment, the experiment will proceed to the next step.

4.2 Code Regeneration

This study used code re-generation to explore and evaluate the accuracy and validity of the code explanations generated in Copilot Lab. The code explanations generated in Copilot Lab needed to be copied into IDEA and the code was re-generated using the installed Copilot plug-in in IDEA. The copying process needed to be done accurately and carefully to ensure that no details were missed. During the experiments, the Copilot regenerated code was too similar to the original code. In order to prevent Copilot from gaining context and creating an inherent bias in the same project or IDE that could affect the experimental results, this study decided to use a different IDE when generating the code.

After copying the code explanations into IDEA, Copilot needs to wait for some time to generate the new code, and although usually the process is quick and seamless, sometimes the waiting time may be too long. When the waiting time is too long, causing Copilot to get stuck or abnormal, IDEA needs to be restarted and the file just opened to regenerate the code. For example, Figure 9 shows the case of Copilot generating code abnormally in IDEA.

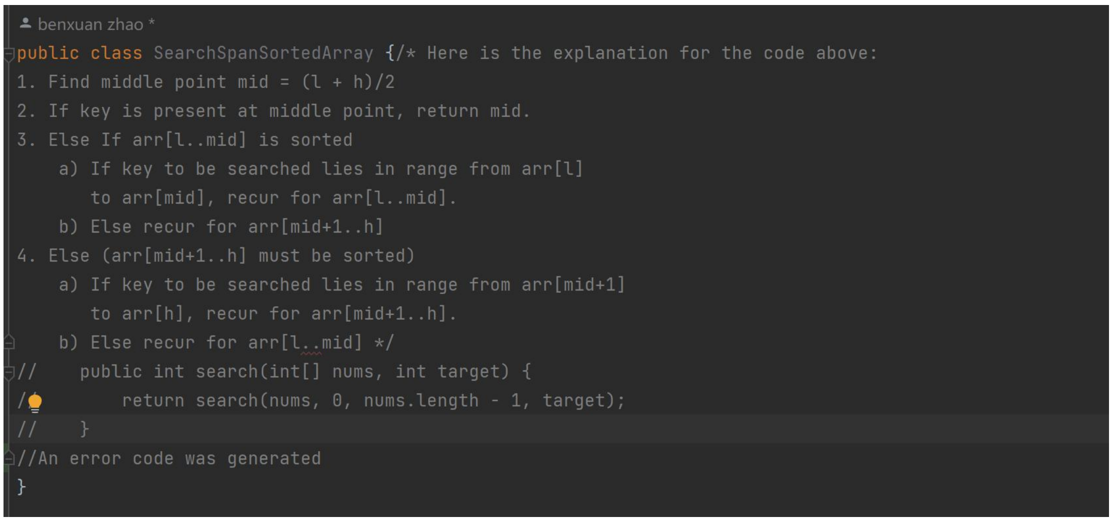
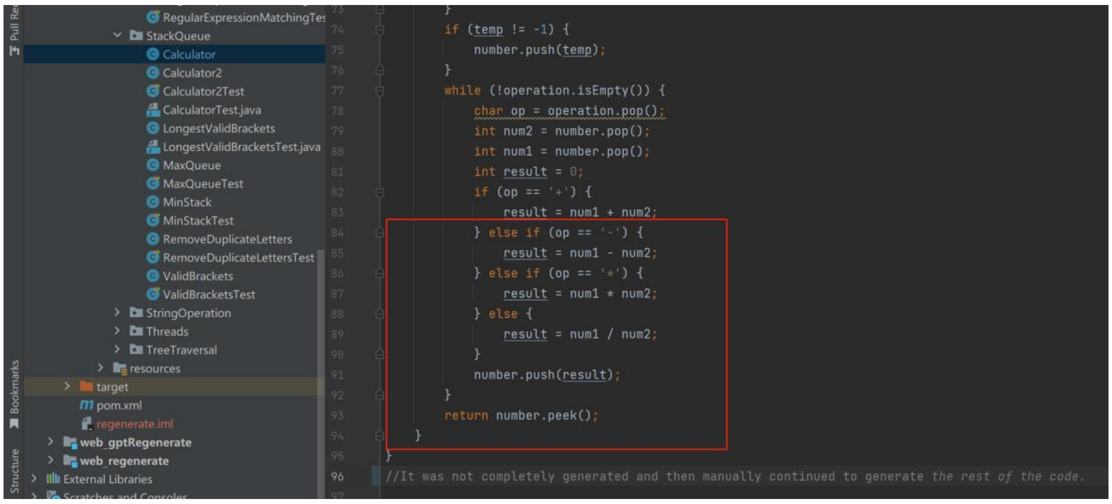

```
benxuan zhao *
public class SearchSpanSortedArray {/* Here is the explanation for the code above:
1. Find middle point mid = (l + h)/2
2. If key is present at middle point, return mid.
3. Else If arr[l..mid] is sorted
    a) If key to be searched lies in range from arr[l]
       to arr[mid], recur for arr[l..mid].
    b) Else recur for arr[mid+1..h]
4. Else (arr[mid+1..h] must be sorted)
    a) If key to be searched lies in range from arr[mid+1]
       to arr[h], recur for arr[mid+1..h].
    b) Else recur for arr[l..mid] */
//    public int search(int[] nums, int target) {
/         return search(nums, 0, nums.length - 1, target);
//    }
//An error code was generated
}
```

Figure 9: Regenerate Code Error

Copilot, although powerful, may still generate incomplete or inaccurate code. Therefore, manual supervision and intervention are crucial in the experimental process. As in Figure 10, Copilot generated only part of the code, and the red box shows the missing part, resulting in an error in the file, which required the experimenter to manually move the cursor to the appropriate position, as shown in the picture example, to the next line, and use Copilot to generate the rest of the code.



```
        }
        if (temp != -1) {
            number.push(temp);
        }
        while (!operation.isEmpty()) {
            char op = operation.pop();
            int num2 = number.pop();
            int num1 = number.pop();
            int result = 0;
            if (op == '+') {
                result = num1 + num2;
            } else if (op == '-') {
                result = num1 - num2;
            } else if (op == '*') {
                result = num1 * num2;
            } else {
                result = num1 / num2;
            }
            number.push(result);
        }
        return number.peek();
    }
}
//It was not completely generated and then manually continued to generate the rest of the code.
```

Figure 10: Regenerate Code Error2

Web projects, on the other hand, are relatively difficult to generate code for again.

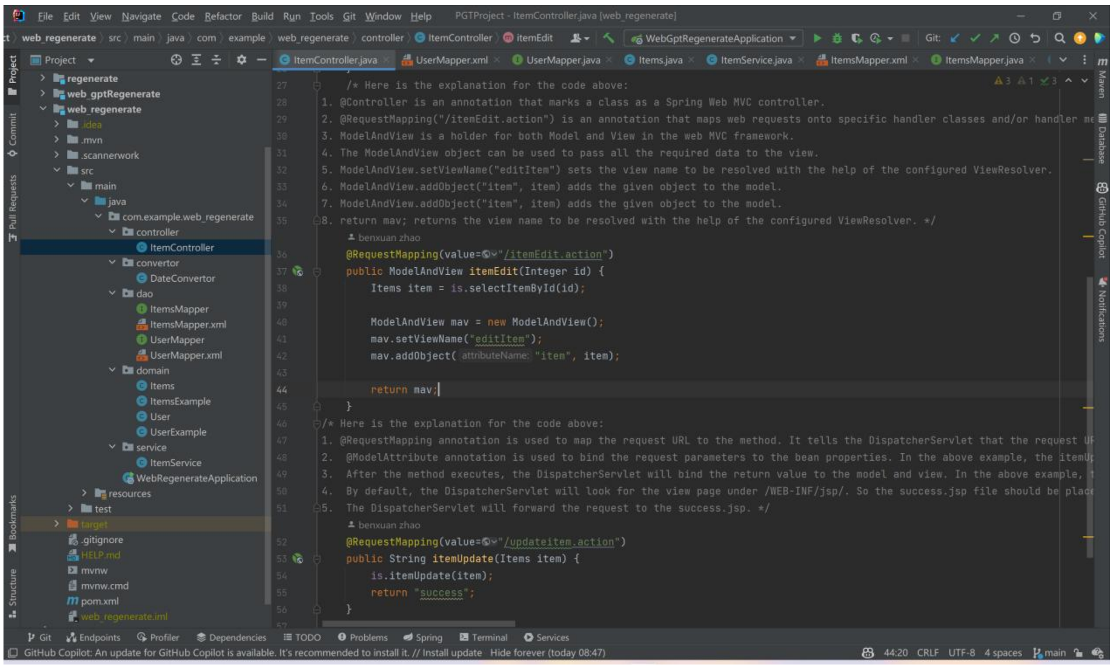
First of all, Copilot cannot successfully generate code using explanations that are generated successfully at class-level granularity (regardless of whether the logic is correct or not), starting with the controller classes, after copying them into IDEA. With correctly logical explanations, Copilot gives no hints after responding, unless the last part of the function is given as a context. For wrong logic explanations, Copilot continues to generate comments. With the alternate solution, Chatgpt4.0 can generate the code directly, but it is not complete due to incomplete code explanation.

Chatgpt4.0 can generate the code directly, but it is not complete due to incomplete code explanation of the service class, but sometimes the generated explanation can be manually replaced by a new line to continue to generate part of the code to continue to generate the code to continue to generate the code can be generated after the complete code.Chatgpt4.0 can generate the code directly, but due to the lack of context, the code can not be used directly, and the class name and the name of the package need to be adjusted manually.

Cannot generate xml headers based on code explanation Unable to fully interpret all properties of resultmap. Copilot can't generate mybatis code based on code explanation. Chatgpt4.0 can but the mapper interface class is reporting some errors, tell gpt the error message and gpt can correct it. But it can generate the following template based on the mybatis template code above.

Cannot generate entity class content based on code explanation.

Copilot can then generate code using the explanations generated at function-level granularity, after copying them into IDEA. An example of successfully generating code with explanations at function-level granularity is shown in Figure 11.



```
    /* Here is the explanation for the code above:
1. @Controller is an annotation that marks a class as a Spring Web MVC controller.
2. @RequestMapping("/itemEdit.action") is an annotation that maps web requests onto specific handler classes and/or handler me
3. ModelAndView is a holder for both Model and View in the web MVC framework.
4. The ModelAndView object can be used to pass all the required data to the view.
5. ModelAndView.setViewName("editItem") sets the view name to be resolved with the help of the configured ViewResolver.
6. ModelAndView.addObject("item", item) adds the given object to the model.
7. ModelAndView.addObject("item", item) adds the given object to the model.
8. return mav; returns the view name to be resolved with the help of the configured ViewResolver. */
    @RequestMapping(value="/itemEdit.action")
    public ModelAndView itemEdit(Integer id) {
        Items item = is.selectItemById(id);

        ModelAndView mav = new ModelAndView();
        mav.setViewName("editItem");
        mav.addObject( attributeName: "item", item);

        return mav;
    }
/* Here is the explanation for the code above:
1. @RequestMapping annotation is used to map the request URL to the method. It tells the DispatcherServlet that the request UR
2.  @ModelAttribute annotation is used to bind the request parameters to the bean properties. In the above example, the itemUp
3.  After the method executes, the DispatcherServlet will bind the return value to the model and view. In the above example, t
4.  By default, the DispatcherServlet will look for the view page under /WEB-INF/jsp/. So the success.jsp file should be place
5.  The DispatcherServlet will forward the request to the success.jsp. */
    @RequestMapping(value="/updateitem.action")
    public String itemUpdate(Items item) {
        is.itemUpdate(item);
        return "success";
    }
```

Figure 11: Regenerate Controller by Copilot

As you can see, Copilot Lab is able to generate explanations for simpler algorithms without any problems and most of them can be used directly. However, for more complex code, the generated explanations are difficult to use. Although the correct explanations are human-readable, they do not strictly define the idea of the code.

Chapter 5 Evaluation

This section describes the analysis of the data obtained from the above experiments in this study.

5.1 Interpretative Quality Analysis

The first step was to analyse the code explanations generated by Copilot Lab in terms of linguistic quality. These explanations were copied line by line to ChatGPT as a language model for evaluation, allowing it to state the strengths and weaknesses of the explanations.

ChatGPT's responses follow the format below:

The function of this text description is:

Advantages are:

Cons are:

All evaluations are recorded in the document and keyword counted, and the strengths are categorised as Clarity/Clear, Detail/Detailed, Logic/Logical, Concise/Conciseness, Simplicity/Simple, Step-by-step/Sequential. Comprehensive, Explanation/Explains, Structure/Structured, Novel/Novelty, Accuracy/Accurate, Explicit/Explicitly, Validity/Valid. Thorough/Thoroughly, Mathematical/Mathematics and Emphasis/Emphasizes for a total of 16.

The disadvantages are classified as Lack of detail/Detail missing, Lack of context, No actual code examples, Too complex for inexperienced programmers, Lack of background information, No explanation provided for the chosen approach, No background information on code design and algorithm selection, A priori knowledge required, etc. There are 23 types in total.

Since there are so many types of drawbacks and they are scattered, several major drawbacks are shown in the frequency histograms plotted to ensure clarity of visualisation.

These records are used as a dataset to generate word clouds and frequency histograms as in Figures 12-15.

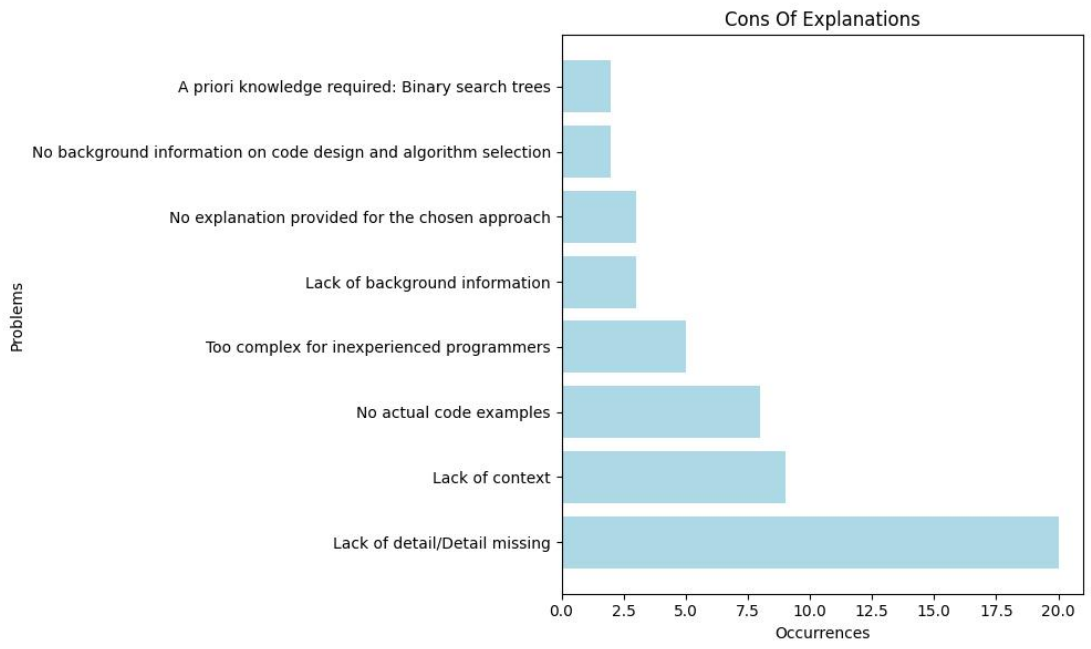


Figure 12: Histogram of Cons

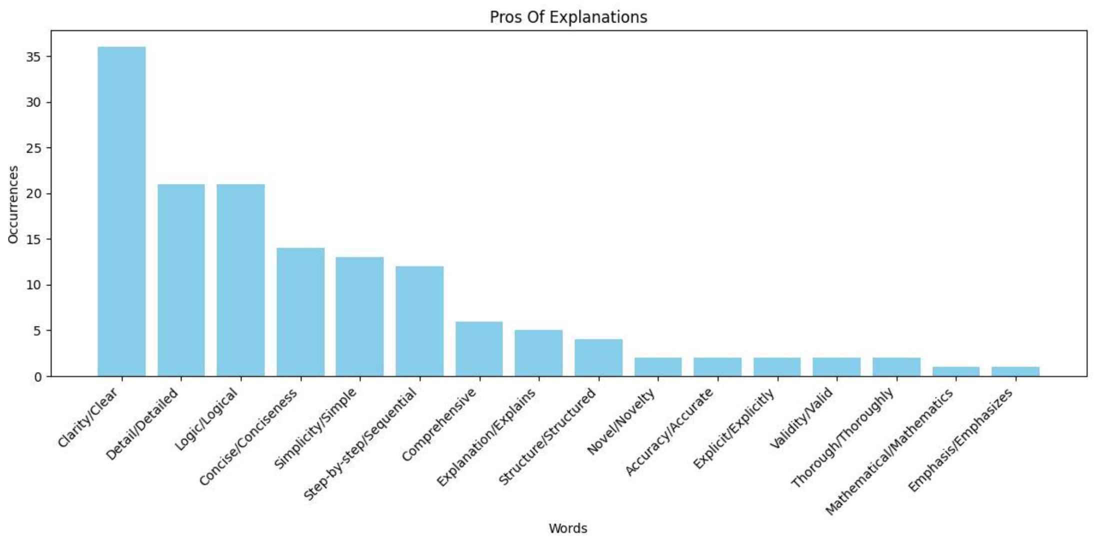


Figure 13: Histogram of Pros

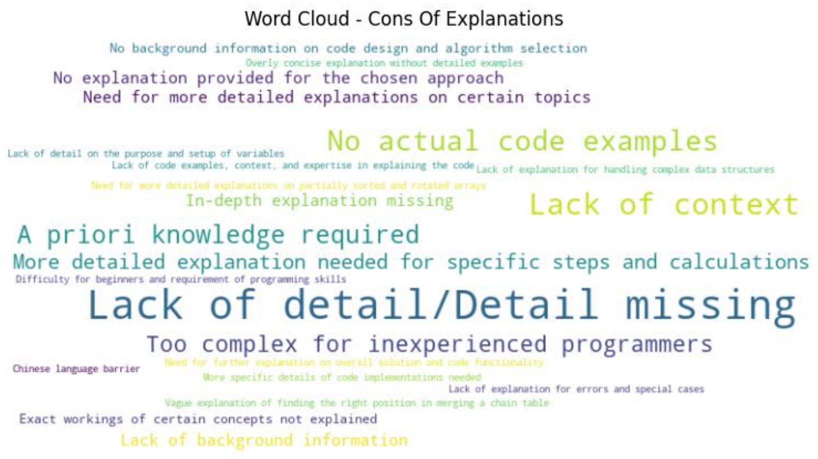


Figure 14: Wordcloud of Cons

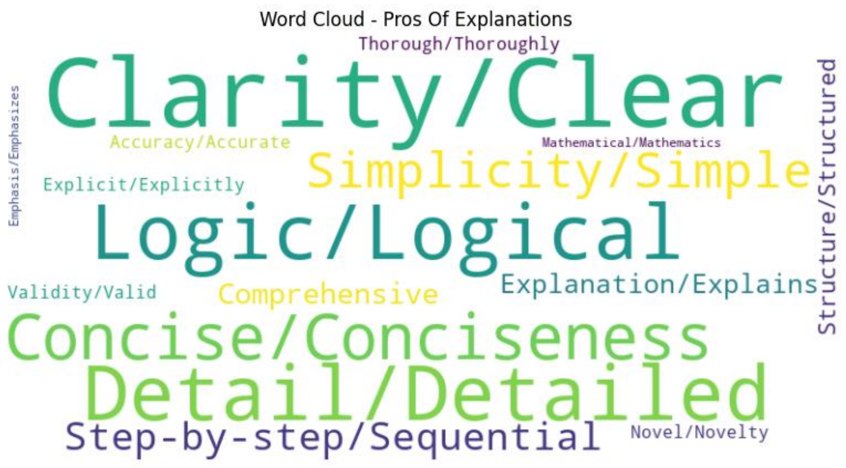


Figure 15: Wordcloud of Pros

As you can see, the explanations generated by Copilot Lab are still very concise, but this also means that some of the explanations will be too simple resulting in a loss of detail. Another problem is that the explanations it generates tend to lack context. Of course, developers may get the context from projects or actual scenarios, but this may still cause the code to be different due to differences in the developer's understanding.

In this process, the analysis results given by ChatGPT were found to be slightly duplicated, so in order to compare and confirm the appropriateness of ChatGPT's evaluation of the explanations, five experimental participants were approached in this experiment to manually evaluate the 10 segments of the explanations that were sampled using the systematic sampling method. The statistical results are shown in Figures 16-17 below.

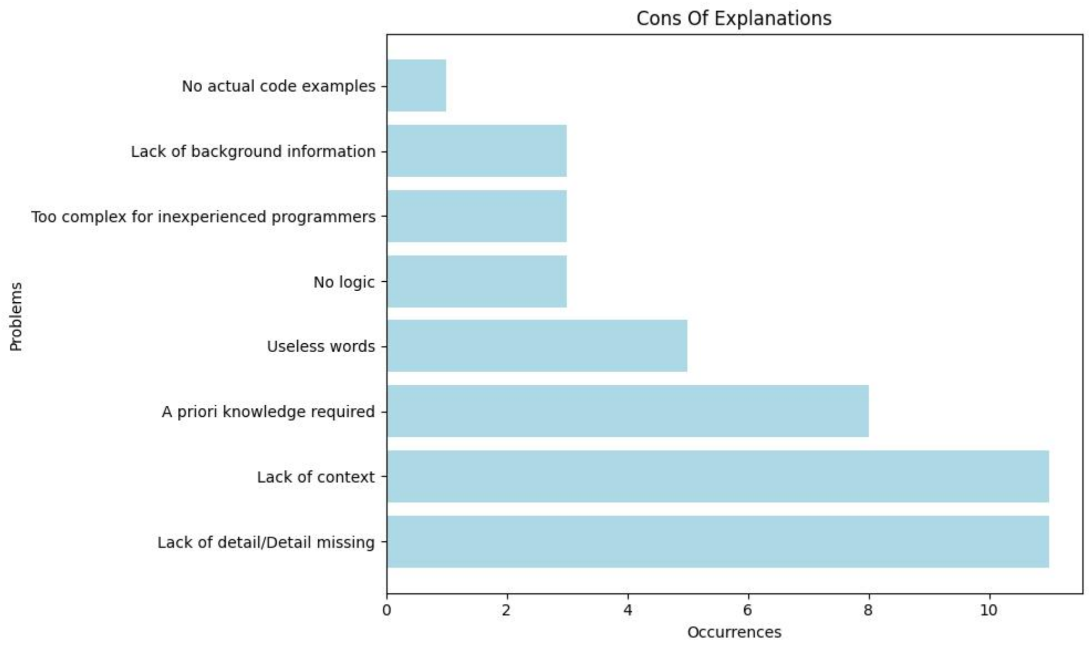


Figure 16: Manually Assessed Bar Chart Of Pros

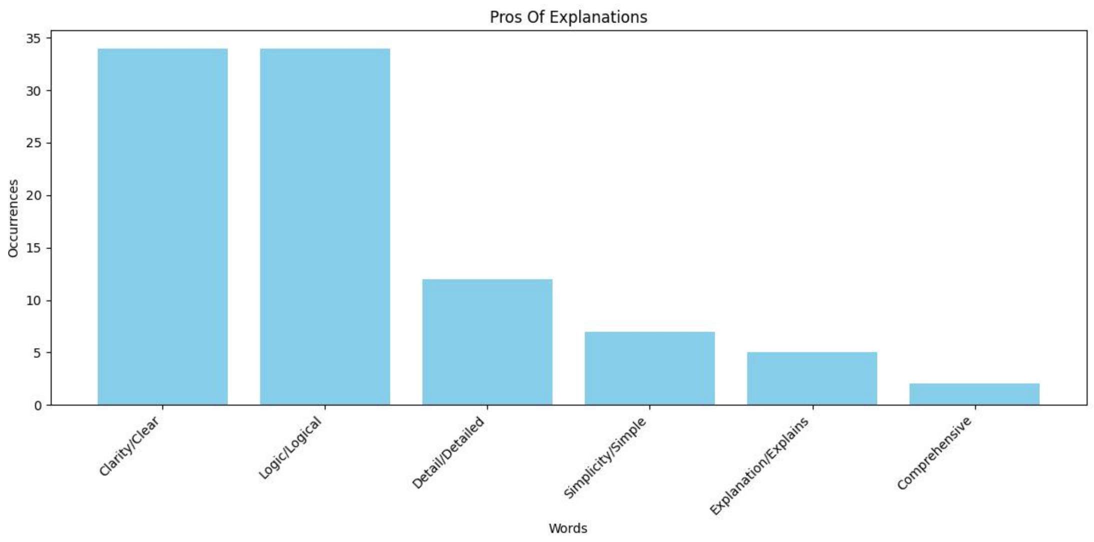


Figure 17: Manually Assessed Bar Chart Of Pros

It is clear that the results of the manual evaluation are the same at least in terms of the most prominent strengths and weaknesses, although there is a considerable difference with ChatGPT. Due to the time constraints and the small sample selected, it was not possible to give an evaluation and statistics for each explanation as in the case of the ChatGPT evaluation, but the manual evaluation proved that the ChatGPT evaluation was generally correct although there were some slight problems.

5.2 Analysis Of The Correlation Between Explanation And Re-generation Of Code

This subsection analyses the Pearson correlation coefficients for several variables mentioned in 3.6, and the appendix shows the time-space complexity of each algorithm.

As Figure 24 shows the visualisation of this analysis, it can be seen that the final degree of code similarity (in descending order) is negatively correlated with the five variables of the quality of the evaluation annotations, i.e. the higher the quality of the evaluation annotations, the higher the degree of code similarity. This proves that the next comparison of the evaluation data is meaningful. It can also be seen that there is a possible negative correlation between time-space complexity and the five variables for assessing annotation quality. This means that the higher the complexity of the algorithmic code, the quality of the code explanation may also be slightly lower.

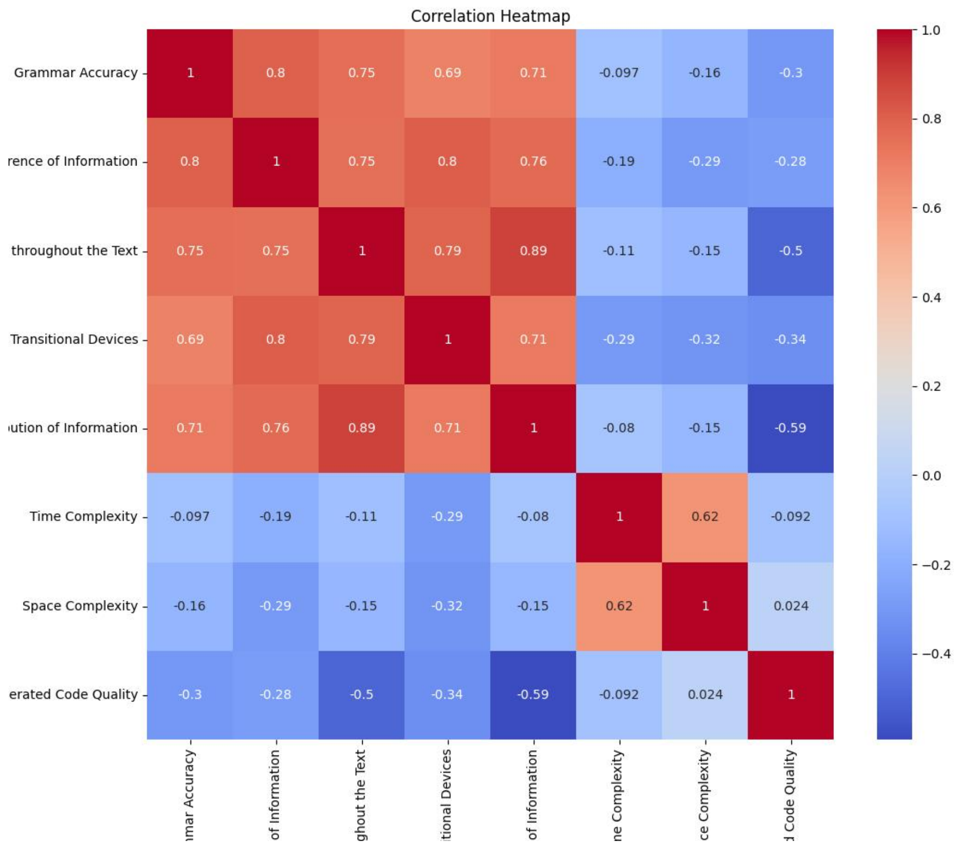
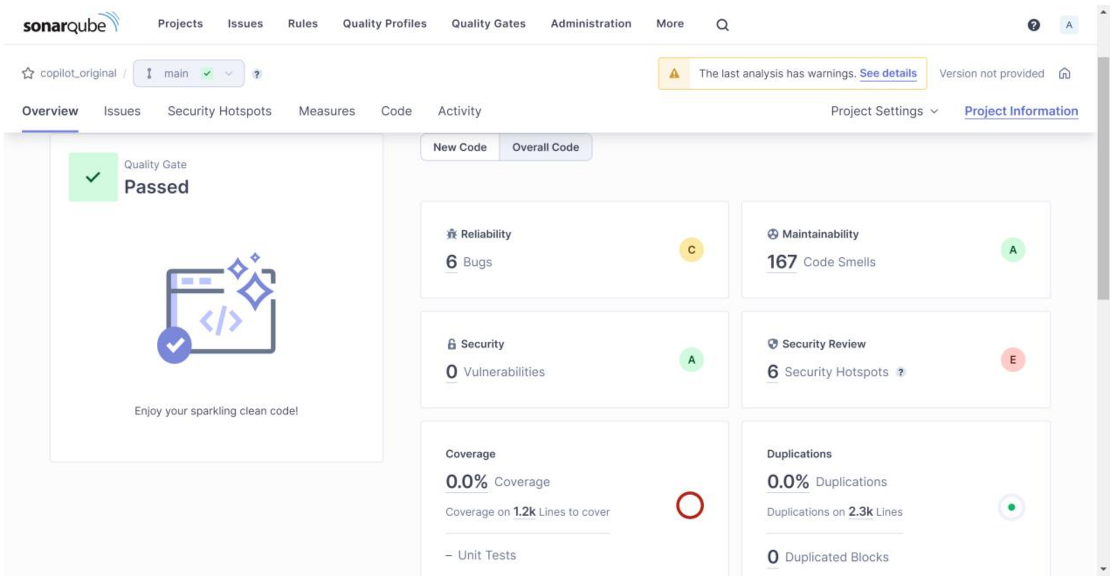


Figure 16: Pearson correlation coefficient Analysis

5.3 Code Quality Analysis

The Leetcode algorithm section, analysed using SonarQube, is shown in Figures 16, 17 below.



The last analysis has warnings. See details

Quality Gate
Passed

Enjoy your sparkling clean code!

Reliability
6 Bugs C

Maintainability
167 Code Smells A

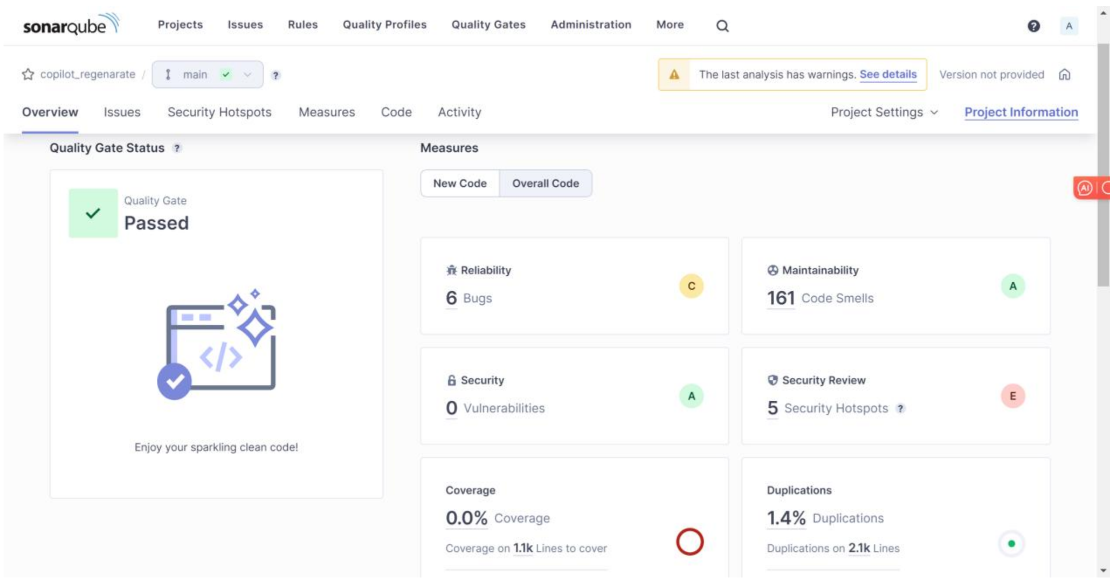
Security
0 Vulnerabilities A

Security Review
6 Security Hotspots E

Coverage
0.0% Coverage
Coverage on 1.2k Lines to cover
– Unit Tests

Duplications
0.0% Duplications
Duplications on 2.3k Lines
0 Duplicated Blocks

Figure 17: Original Leetcode Project Analysis


sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration More

copilot_regenarate / main

The last analysis has warnings. See details Version not provided

Overview Issues Security Hotspots Measures Code Activity Project Settings Project Information

Quality Gate Status

Quality Gate Passed

Enjoy your sparkling clean code!

Measures

New Code Overall Code

Reliability: 6 Bugs — C

Maintainability: 161 Code Smells — A

Security: 0 Vulnerabilities — A

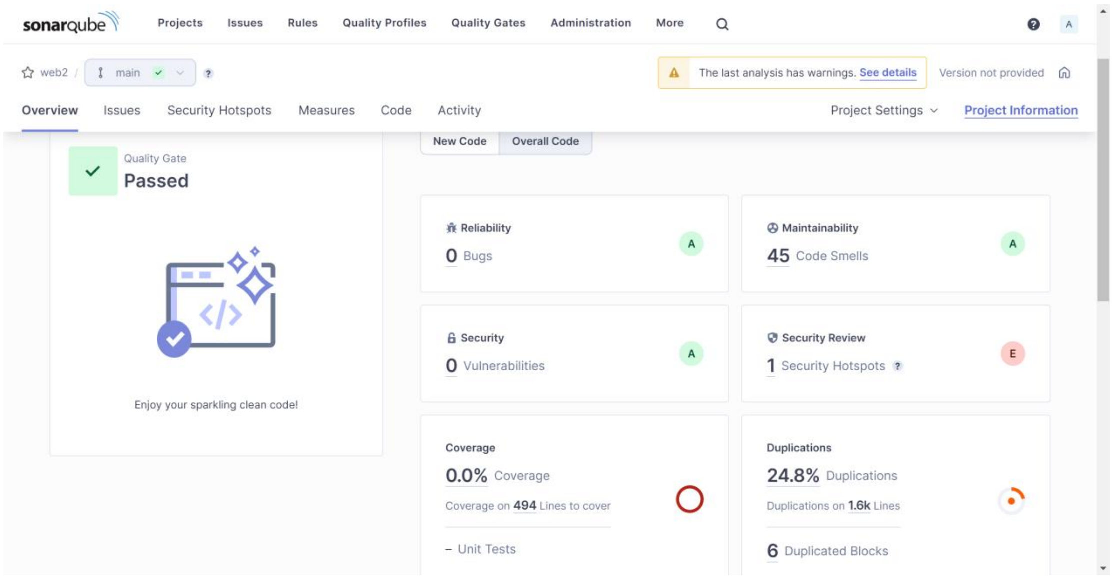
Security Review: 5 Security Hotspots — E

Coverage: 0.0% Coverage — Coverage on 1.1k Lines to cover

Duplications: 1.4% Duplications — Duplications on 2.1k Lines

Figure 18: Regenerated Leetcode Project Analysis

For the web project part, since the code was generated using ChatGPT heavily at a large granularity during the experiment, a separate project comparison was simply created for this experiment to generate the analysis results using SonarQube as shown in Figures 19-21 below.


sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration More

web2 / main

The last analysis has warnings. See details Version not provided

Overview Issues Security Hotspots Measures Code Activity Project Settings Project Information

New Code Overall Code

Quality Gate Passed

Enjoy your sparkling clean code!

Reliability: 0 Bugs — A

Maintainability: 45 Code Smells — A

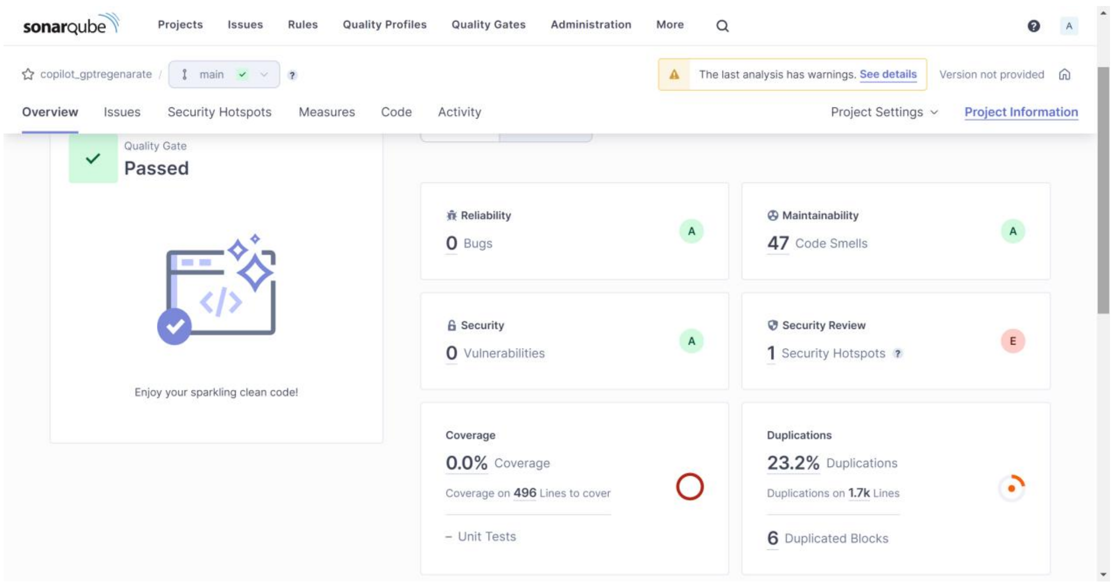
Security: 0 Vulnerabilities — A

Security Review: 1 Security Hotspots — E

Coverage: 0.0% Coverage — Coverage on 494 Lines to cover — Unit Tests

Duplications: 24.8% Duplications — Duplications on 1.6k Lines — 6 Duplicated Blocks

Figure 19: Original Web Project Analysis


sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration More

copilot_gptregenarate / main

The last analysis has warnings. See details

Version not provided

Overview Issues Security Hotspots Measures Code Activity Project Settings Project Information

Quality Gate
Passed

Enjoy your sparkling clean code!

Reliability
0 Bugs
A

Maintainability
47 Code Smells
A

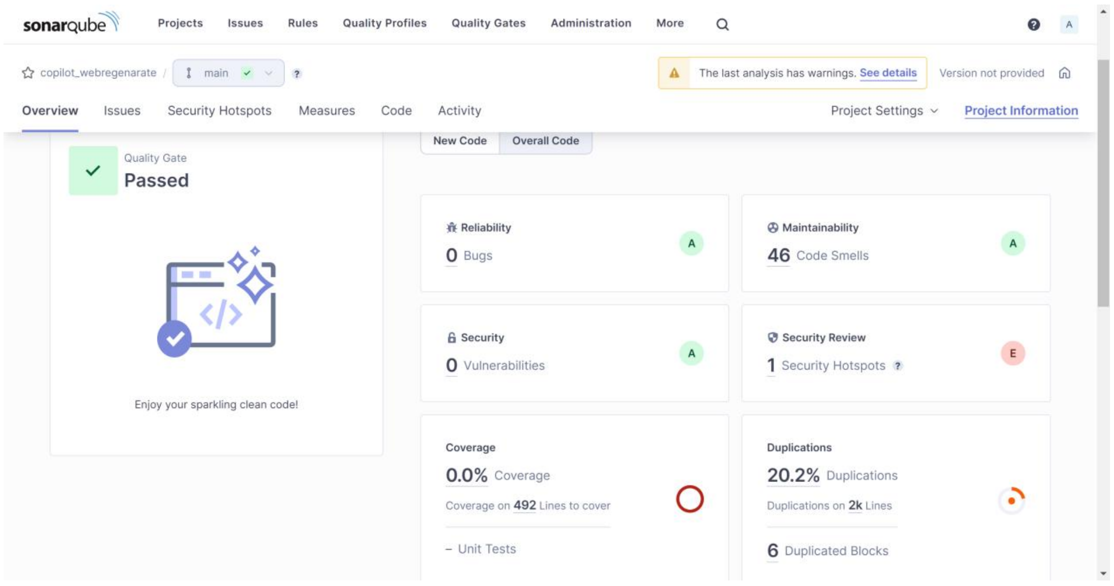
Security
0 Vulnerabilities
A

Security Review
1 Security Hotspots
E

Coverage
0.0% Coverage
Coverage on 496 Lines to cover
– Unit Tests

Duplications
23.2% Duplications
Duplications on 1.7k Lines
6 Duplicated Blocks

Figure 20: Copilot Regenerated Web Project Analysis


sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration More

copilot_webregenarate / main

The last analysis has warnings. See details

Version not provided

Overview Issues Security Hotspots Measures Code Activity Project Settings Project Information

New Code Overall Code

Quality Gate
Passed

Enjoy your sparkling clean code!

Reliability
0 Bugs
A

Maintainability
46 Code Smells
A

Security
0 Vulnerabilities
A

Security Review
1 Security Hotspots
E

Coverage
0.0% Coverage
Coverage on 492 Lines to cover
– Unit Tests

Duplications
20.2% Duplications
Duplications on 2k Lines
6 Duplicated Blocks

Figure 21: ChatGPT Regenerated Web Project Analysis

It can be seen that the difference in quality between the code generated using Copilot Lab's annotations and the source code is not significant for both algorithmic and web projects. However, it is worth noting that the generated algorithmic code shows a decrease in code smells for Maintainability but an increase in Duplications, while the opposite is true for web code. This may be due to the fact that leetcode algorithmic questions have much the same answers on github, whereas web projects are complex and varied, and are prone to be trained to different code styles, which include bad code usage.

As web project code is not successfully generated by most Copilots, even using ChatGPT 4.0 it is quite dependent on the given context within the same file and often reports errors. Here it was not considered possible to use AI to generate web projects based on code explanation. Therefore, there is no need to carry out the following code variance analysis and performance analysis.

5.4 Code Variance Analysis

As shown in Figure 22, a visualisation of the data obtained from the comparison of the original and re-generated codes by the participants of this study is divided into 16 groups on the x-axis according to the type of algorithm and the data structure, with the bar on the left side of each group being the degree of similarity of the codes, and the bar on the right side being the degree of superiority of the two codes in the case of the dissimilar codes.

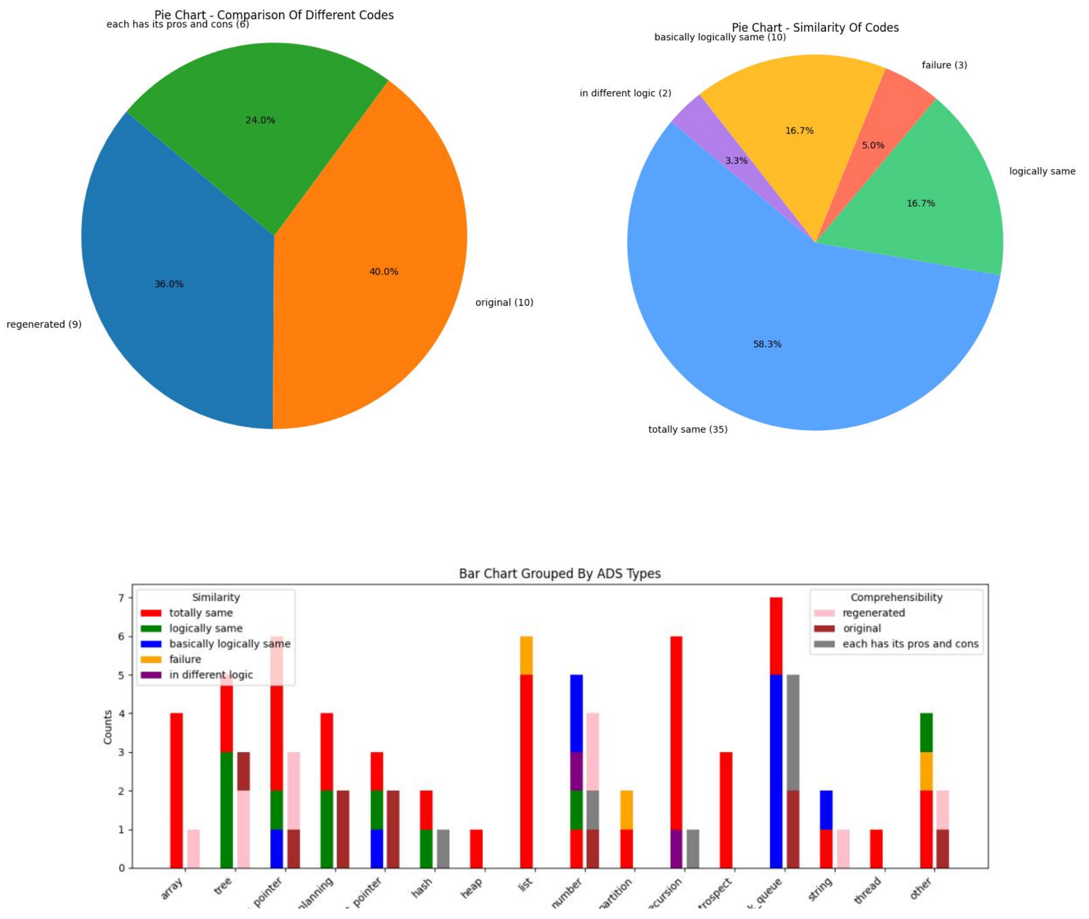


Figure 22: ChatGPT Regenerated Web Project Analysis

It can be seen that the re-generated code does not differ much in quality from the original code in general and is mostly identical, but is of higher quality in the application of the underlying data structures and not as good as the original code in terms of algorithms.

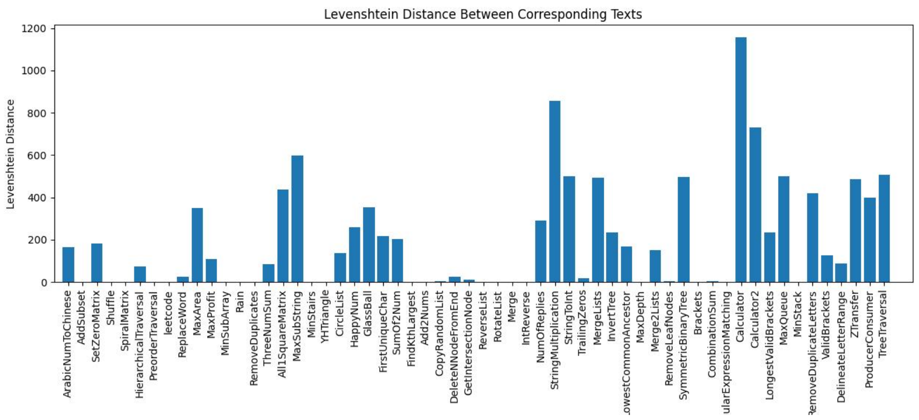


Figure 23: ChatGPT Regenerated Web Project Analysis

As can be seen in Figure 22, which shows the Levenshtein distance between the original code and the regenerated code for each algorithm, with the same type of algorithms in close proximity to each other, the graph exhibits a trend, i.e., the magnitude of the increase or decrease in each of the bars from left to right, which is roughly the same as the trend in the degree of similarity of the code as perceived by the study participants, i.e., the magnitude of the increase or decrease in the height of the bar on the right side of each group in Figure 21.

From this, it can be determined that the re-generated code does not differ much from the quality of the original code in general, either subjectively or objectively by the developers.

5.5 Code Performance Analysis

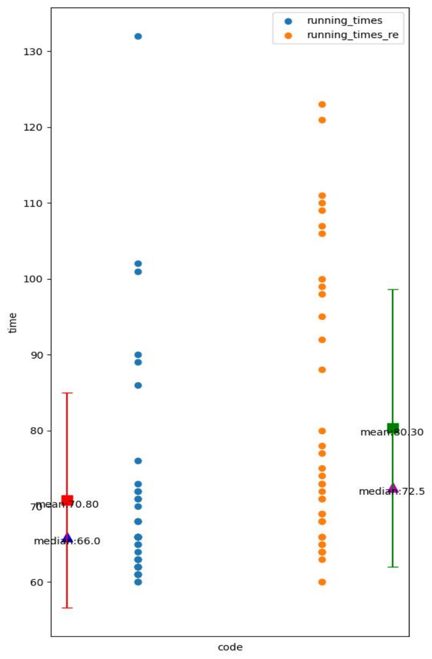


Figure 24: JUnit Running Times

As Figure 24 shows the statistical results of the overall running time obtained from 40 unit tests of all methods of the Data Structures and Algorithms project using JUnit, the blue dots are the data of the original code, the orange dots are the data in the regenerated code, and the median, mean and standard deviation of the corresponding code are on both sides, respectively.

It can be seen that the median mean and standard deviation of the overall running time of the original code are smaller than those of the regenerated code, so it can be assumed that the overall running efficiency of the regenerated code is inferior to that of the original code.

The Web project is not subject to this analysis because some of the code could not be generated.

Chapter 6 Conclusion

This study assesses the quality of Copilot Lab code explanations through proxy measurements, a unique method for assessing explanation accuracy. The experiment compares the original code with the re-generated code and analyses the differences in their quality, performance and the code itself.

The results show that in the data structures and algorithms project, there are some differences between the re-generated code and the original code, mainly in terms of performance and the data structures used, and that the re-generated code is comparable to the original code in terms of quality. In Web projects, it is quite difficult to regenerate codes by explanations.

This suggests that for simple code, Copilot Lab generates good explanations, but for more complex code, it will not be able to generate suitable explanations to help people develop code or write documentation.

6.1 Future Work

The code generated using ai in this study has many unknowns and does not fully simulate the conditions that people work under. And the direct assessment of explanations was not rigorous enough.

In the future the code generation and direct assessment of the explanations can be done using expert and user surveys to make the experiments more realistic, and the quality of the generated explanations can be assessed using methods from the NLP field such as BLEU.

Additionally a larger number of code samples could be collected, refining the number of samples per project and algorithm category to make the experiments more generalisable. It is even possible to use manually written code samples on an ad hoc basis, thus avoiding duplication of these codes as a test set with Copilot's and Copilot Lab's training sets on the web.

Appendix A Complexities Of Arithmetic

Classes	Time Complexity	Space Cpmplexity	Type
ArabicNumToChinese	O(log(N))	O(log(N))	Other
AddSubset	O(N * 2^N)	O(N * 2^N)	Array
SetZeroMatrix	O(M * N)	O(1)	Array
Shuffle	O(N^2)	O(N)	Array
SpiralMatrix	O(m * n)	O(m * n)	Array
HierarchicalTraversal	O(N)	O(max(N, W))	Tree
PreorderTraversal	O(N)	O(max(N, H))	Tree
DeleteBTSNode	O(n)	O(n)	Tree
ReplaceWord	O(n + m * k)	O(n + k)	Tree
Dijkstra	O(n^2)	O(n^2)	Other
MaxArea	O(n)	O(1)	DoublePointer
MaxProfit	O(n)	O(1)	DoublePointer
MinSubArray	O(n)	O(1)	DoublePointer
Rain	O(n^2)	O(1)	DoublePointer
RemoveDuplicate	O(n)	O(1)	DoublePointer
ThreeNumSum	O(n^2)	O(n^2)	DoublePointer
All1SquareMatrix	O(m * n)	O(m * n)	DynamicPlanning
MaxSubString	O(m * n)	O(m * n)	DynamicPlanning
MinStairs	O(n)	O(1)	DynamicPlanning
YHTriangle	O(numRows^2)	O(numRows^2)	DynamicPlanning
CircleList	O(n)	O(1)	FastAndSlowPointer
HappyNum	O(log n)	O(1)	FastAndSlowPointer

GlassBall	O(n^2)	O(n^2)	Other
FirstUniqueChar	O(n)	O(n)	Hash
SumOf2Num	O(n)	O(n)	Hash
FindKthLargest	O(n log k)	O(k)	Heap
Add2Nums	O(max(m, n))	O(max(m, n))	List
CopyRandomList	O(n)	O(n)	List
DeleteNNodeFromEnd	O(n)	O(1)	List
GetIntersectionNode	O(m+n)	O(m)	List
ReverseList	O(n)	O(1)	List
RotateList	O(n)	O(1)	List
Merge	O(nlogn)	O(n)	Array
IntReverse	O(log|x|)	O(1)	Number
NumOfReplies	O(log|x|)	O(1)	Number
StringMultiplication	O(m * n)	O(m + n)	Number
StringToInt	O(n)	O(1)	Number
TrailingZeros	O(logn)	O(1)	Number
MergeLists	O(m+n)	O(1)	Partitioning
SearchSpanSortedArray	O(log n)	O(log n)	Partitioning
InvertTree	O(n)	O(log(n))	Recursion
LowestCommonAncestor	O(n)	O(log(n))	Recursion
MaxDepth	O(n)	O(log(n))	Recursion
Merge2Lists	O(m+n)	O(m+n)	Recursion
RemoveLeafNodes	O(n)	O(n)	Recursion
SymmetricBinaryTree	O(n)	O(n)	Recursion
Brackets	O(Catalan(n))	O(n)	Retrospect
CombinationSum	O(2^n)	O(n)	Retrospect
RegularExpressionMatchi	O(m * n)	O(m + n)	Retrospect

ng			
Calculator	O(n)	O(n)	StackAndQueue
Calculator2	O(n)	O(n)	StackAndQueue
LongestValidBrackets	O(n)	O(n)	StackAndQueue
MaxQueue	O(k)	O(n)	StackAndQueue
MinStack	O(1)	O(n)	StackAndQueue
RemoveDuplicateLetters	O(n)	O(n)	StackAndQueue
ValidBrackets	O(n)	O(n)	StackAndQueue
DelineateLetterRange	O(n)	O(n)	String
ZTransfer	O(n + numRows * m)	O(n)	String
ProducerConsumer	multi-threaded	multi-threaded	multi-threaded
TreeTraversal	O(n)	O(n)	Tree

Bibliography

[1] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '22). Association for Computing Machinery, New York, NY, USA, 1019 – 1027. https://doi.org/10.1145/3512290.3528700

[2] Naser Al Madi. 2023. How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 205, 1 – 5. https://doi.org/10.1145/3551349.3560438

[3] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking Flight with Copilot. Commun. ACM 66, 6 (June 2023), 56 – 62. https://doi.org/10.1145/3589996

[4] Yukinao Hirata and Osamu Mizuno. 2011. Do comments explain codes adequately? investigation by text filtering. In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11). Association for Computing Machinery, New York, NY, USA, 242 – 245. https://doi.org/10.1145/1985441.1985482

[5] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students Struggle to Explain Their Own Program Code. In Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE '21). Association for Computing Machinery, New York, NY, USA, 206 – 212. https://doi.org/10.1145/3430665.3456322

[6] Donna Harman. 2003. Issues in reuse of evaluation data and metrics. In Proceedings of the EACL 2003 Workshop on Evaluation Initiatives in Natural Language Processing: are evaluation methods, metrics and resources reusable? (Evalinitiatives '03). Association for Computational Linguistics, USA, 1.

[7] 2003. Proceedings of the EACL 2003 Workshop on Evaluation Initiatives in Natural Language Processing: are evaluation methods, metrics and resources reusable? Association for Computational Linguistics, USA.

[8] Lynette Hirschman. 1990. Session details: Natural language evaluation. In Proceedings of the workshop on Speech and Natural Language (HLT '90). Association for Computational Linguistics, USA.

[9] Rita McCardell. 1990. Evaluating natural language generated database records. In Proceedings of the workshop on Speech and Natural Language (HLT '90). Association for Computational Linguistics, USA, 64 – 70. https://doi.org/10.3115/116580.116607

[10] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22). Association for Computing Machinery, New York, NY, USA, 1 – 5. https://doi.org/10.1145/3524842.3528470

[11] Simon, Oscar Karnalim, Judy Sheard, Ilir Dema, Amey Karkare, Juho Leinonen, Michael Liut, and Renee McCauley. 2020. Selection of Code Segments for Exclusion from Code Similarity Detection. In Proceedings of the 2020 ACM Conference on Innovation and

Technology in Computer Science Education (ITiCSE '20). Association for Computing Machinery, New York, NY, USA, 500 – 501. https://doi.org/10.1145/3341525.3394987

[12] Saki Imai. 2022. Is GitHub copilot a substitute for human pair-programming? an empirical study. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22). Association for Computing Machinery, New York, NY, USA, 319 – 321. https://doi.org/10.1145/3510454.3522684

[13] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot ' s code generation. In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2022). Association for Computing Machinery, New York, NY, USA, 62 – 71. https://doi.org/10.1145/3558489.3559072

[14] https://githubnext.com/projects/copilot-labs/#explain-this-code

[15] https://platform.openai.com/docs/guides/gpt-best-practices/strategy-write-clear-instructions

[16] https://github.com/ruanwenjun/JAVAWeb-Project/tree/master/SSM%E7%AE%80%E5%8D%95%E6%95%B4%E5%90%88

[17] https://github.com/yuanguangxin/LeetCode/blob/master/src/hash%E7%9B%B8%E5%85%B3/q1_%E4%B8%A4%E6%95%B0%E4%B9%8B%E5%92%8C/f1/Solution.java

[18] Halliday, M. A. K., & Hasan, R. 1976. Cohesion in English. Longman.

[19] Dijk, T. A. v., Dijk, T. A. v. 1977. Text and Context: Explorations in the Semantics and Pragmatics of Discourse. Longman.

[20] The Handbook of Discourse Analysis. 2018. Wiley.

[21] Schiffrin, D. 1988. Discourse Markers. Cambridge University Press.

[22] Riessman, C. K. 2022. Narrative Analysis. SAGE Publications.