An empirical analysis of GitHub Copilot

Isak Nilsson
An empirical analysis of GitHub Copilot

Isak Nilsson

Abstract

The rapid advancement of artificial intelligence (AI) has brought transformative changes to various fields, including software development. GitHub Copilot, an AI-powered code assistant developed by OpenAI in collaboration with Microsoft, has emerged as a promising tool that aims to enhance developers’ productivity by providing context-aware code suggestions. This bachelor’s thesis presents an empirical evaluation of GitHub Copilot, exploring its effectiveness and efficiency.

The study adopts a mixed-methods approach, combining both quantitative and qualitative techniques to assess GitHub Copilot’s performance in aiding software developers with solving work-related programming tasks. Additionally, after solving the tasks, a survey is conducted with the participants to gather their perceptions and overall impression of Copilot.

The results of the empirical evaluation provide evidence-based insights into the effects of using Copilot as a programming tool. Findings from the quantitative analysis reveal Copilot’s potential to significantly reduce development time, while also slightly increasing the code’s quality. Moreover, the qualitative analysis did not show any difference in the code’s efficiency when using Copilot.
## Contents

1 Introduction ................................. 6

2 Background and Terminology ............... 7

3 Related Work ................................ 8

4 Study Design ................................ 9
   4.1 The Tasks .................................. 12
      4.1.1 Task 1 .................................. 12
      4.1.2 Task 2 .................................. 14
   4.2 Pilot Study ................................. 18

5 Results ....................................... 19
   5.1 Quantitative ............................... 19
   5.2 Qualitative ................................. 20
   5.3 Survey Evaluation ......................... 23

6 Discussion ................................... 24

7 Conclusion ................................... 27

A Appendix .................................... 29
1 Introduction

Artificial intelligence (AI) has in recent years made some significant advances in various fields, including software development. One such advancement is the emergence of AI-powered code assistants such as GitHub Copilot [1], a state-of-the-art cloud-based AI system developed by OpenAI and GitHub that uses natural language processing as well as machine learning to assist software developers in writing code by generating code suggestions and completing code snippets. These code suggestions can be based on previous code in the developers’ code base to match its coding convention and can also be generated from comments and function names.

While there are several existing coding assistant tools available, such as IntelliSense and IntelliCode, GitHub Copilot stands out due to its advanced capabilities. Copilot is more powerful because it uses natural language processing, machine learning, and big data to suggest code, allowing it to generate code snippets that match the developer’s coding convention and even complete entire functions. This makes it a promising tool for increasing productivity and improving code quality in the software development industry. However, it is important to evaluate its effectiveness and limitations to fully understand its potential impact. This report aims to do just that, by conducting a user study of eight participants of varying programming experience from the software development company, Frontwalker [1] to investigate the effectiveness and efficiency of GitHub Copilot as a code assistant tool in a workplace environment.

What will not be covered in this report are the legal and ethical aspects of using Copilot which is a complex and wide-ranging topic that requires a dedicated analysis that is beyond the scope of this report. The legal and ethical considerations surrounding Copilot usage are multi-faceted, including concerns about ownership of the generated code, potential privacy issues as well as accountability in case of any negative outcomes. One of the aforementioned potential privacy issues is that in order for Copilot to generate code it has to scan your current code base, and while you may opt out of letting Copilot generate code for other users based on your code, the code base would still be stored in GitHub’s servers, and be shared with both Microsoft and OpenAI [2]. Due to the depth and breadth of these issues, a comprehensive examination of them would require extensive research and discussion, which is orthogonal to the focus of this thesis.

In order to evaluate GitHub Copilot as a code assistant tool a user study was conveyed involving two completely different, work-related programming tasks. The study focused on experienced developers working within software develop-

---

1 https://frontwalker.se/om-oss/
ment at Frontwalker where each participant had to solve one of the tasks using GitHub Copilot and the other without Copilot.

The findings of this study suggest that participants using Copilot both completed their tasks faster and had a higher success rate in passing the tests than those not using Copilot.

2 Background and Terminology

Currently, there are many different kinds of code completion and code suggestion tools available for software developers, such as IntelliSense and IntelliCode which are also developed by Microsoft.

*IntelliSense* is a code completion tool that suggests code snippets as developers write code based on the current context of the code, including variable names, functions, and parameters. IntelliSense is limited to the code within the project or the libraries used in the project. According to the Microsoft documentation, IntelliSense works by analyzing code in the background as you write and then provides suggestions for classes, methods, and functions as well as their parameters and return types. It can also suggest appropriate code snippets and provide information on code errors or warnings.

*IntelliCode* is an extension of IntelliSense that uses machine learning models to provide code completion and suggestions. It goes beyond the simple code completion provided by IntelliSense by analyzing code patterns and providing more intelligent suggestions. IntelliCode is trained on a data set of thousands of open-source projects and can suggest entire code blocks and functions based on the context of the code being written. GitHub Copilot is an AI-powered code completion tool developed by OpenAI in collaboration with Microsoft. It was released in June 2021 and has generated significant interest in the software development community. GitHub Copilot utilizes large language models, powered by OpenAI’s Codex model, which is a descendent of GPT-3. OpenAI recently garnered increased public attention with its official release of ChatGPT in December 2022, which is also a large language model powered by its GPT-3.5 which has further sparked the public’s interest in the development of artificial intelligence. The Codex model powering Copilot is trained on a vast amount of code from GitHub repositories to provide developers with intelligent suggestions for completing code snippets. It promises to save

---

5 https://www.theguardian.com/technology/2022/dec/05/what-is-ai-chatbot-phenomenon-chatgpt-and-could-it-replace-humans
developers time, allowing them to focus on solving bigger problems and making them feel more fulfilled with their work, and making coding more efficient [1].

GitHub Copilot has three main functionalities: generating code based on comments and function names, generating tests for already implemented code, and auto-fill for repetitive code. This study will focus on the first functionality, generating code based on comments and function names, which is triggered by the user writing a comment describing the functionality and logic of what is to be implemented, or by simply providing a function’s name. It is also recommended that the user provide meaningful function and parameter names as well as descriptive comments to improve the code recommendations.

To summarize, IntelliSense provides basic code completion and code suggestions based on the code being written while IntelliCode uses machine learning to provide more advanced suggestions based on code patterns and context. IntelliCode can also suggest entire blocks of code and functions, not unlike GitHub Copilot. However, GitHub Copilot can be seen as a more advanced version of IntelliSense and IntelliCode as while these tools are limited to suggesting code snippets based on the code being written, Copilot is capable of generating entire code snippets and functions based on natural language provided by the developer [1]. GitHub Copilot also has access to a much larger data set of code, as it is trained on code from public GitHub repositories, whereas IntelliSense is limited to code within the project or the libraries used in the project.

3 Related Work

Research and studies have already been conducted to research GitHub Copilot’s capabilities as a programming tool. However, to the best of my knowledge, no study has been done using the programming language Java where the study participants are currently active software developers.

Previous empirical evaluations of Copilot, such as Nhan Nguyen and Sarah Nadi’s study [3], have focused on Copilot’s raw code suggestions and its readability rather than its use as an enhancement tool for software developers. Moreover, while these studies may have utilized Copilot to solve programming problems, they typically utilized algorithmic problems from sites like LeetCode, which may not reflect the type of work-related programming problems developers typically encounter.

Arghavan et al.’s report "GitHub Copilot AI pair programmer: Asset or Liability?” [4] solely examines the effectiveness of Copilot’s code generation capabilities for fundamental algorithmic problems, comparing its suggested solutions to those written by a human. This approach differs from the focus of this report, which aims to investigate Copilot as a programming tool for software developers to enhance their productivity and efficiency, rather than entirely replacing
the role of the programmer. However, it is worth noting that Arghavan et al.’s report provides valuable insight into Copilot’s capabilities as a code generation tool, demonstrating its potential to generate code solutions that are on par with those written by humans for basic algorithmic problems. This highlights the potential of Copilot to assist developers in writing code more efficiently, freeing up their time to work on more complex programming tasks. This resonates with the focus of this report, which aims to investigate the potential of Copilot as a tool for software developers to improve their productivity and efficiency, rather than replacing them entirely.

Priyan Vaithilingam, Tianyi Zhang, and Elena Glassman also conducted a user study to evaluate how Copilot affects the programming experience in a very similar way to this study [5]. However, they chose to compare Microsoft’s IntelliSense against GitHub’s Copilot whereas this study is comparing IntelliSense against both Copilot and IntelliSense combined. This is because, in my opinion, these two tools are not replacements for each other and a software developer generally uses multiple types of different tools rather than limiting themselves to a specific one. Priyan et al. also focused on how users recognize errors in code generated by Copilot and on what coping mechanisms users employ once they find errors in the code generated by Copilot which is not in the scope of this report. Additionally, Priyan et al. used the programming language Python for their study whereas this study was done in Java, using experienced software developers working at Frontwalker for the user study rather than students.

While there are some similarities between this report and Saki Imai’s report “Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study” [6] such as if Copilot can increase a programmer’s productivity Saki’s report focuses on comparing Copilot as a pair-programmer to a human pair-programmer. Comparing an AI programming tool to an actual software developer is a very ambitious comparison with a lot of variables, such as the experience of said pair-programmer and the relationship dynamics between them. Also, in light of the current state of technology, it would be considered a positive outcome for an AI programming tool to achieve even half the effectiveness of a human pair-programmer. However, existing technology has not yet reached this level of proficiency. This aligns with the results of Saki Imai’s report, which suggests that pair programming with Copilot does not match the performance of human pair programming.

4 Study Design

For this study, the following research questions were investigated:

1. **RQ1:** Will software developers using GitHub Copilot complete their tasks faster than people not using Copilot?

2. **RQ2:** Will code written by software developers using GitHub Copilot be
better and more efficient than by those not using Copilot?

<table>
<thead>
<tr>
<th>Participant</th>
<th>Programming Experience</th>
<th>Java Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5-10 years</td>
<td>1 year</td>
</tr>
<tr>
<td>b</td>
<td>10+ years</td>
<td>10+ years</td>
</tr>
<tr>
<td>c</td>
<td>5-10 years</td>
<td>1 year</td>
</tr>
<tr>
<td>d</td>
<td>1 year</td>
<td>1-3 years</td>
</tr>
<tr>
<td>e</td>
<td>3-5 years</td>
<td>1-3 years</td>
</tr>
<tr>
<td>f</td>
<td>1 year</td>
<td>1 year</td>
</tr>
<tr>
<td>g</td>
<td>10+ years</td>
<td>10+ years</td>
</tr>
<tr>
<td>h</td>
<td>10+ years</td>
<td>5-10 years</td>
</tr>
</tbody>
</table>

Table 1: Participants’ experience in programming, programming experience is for how long they have been working with programming and Java experience is for how long they have been using Java, both for work, studies, or in their free time.

Eight participants of varying programming experience from Frontwalker were recruited for this study, as can be seen in Table 1.

What this study aims to measure is how quickly, correctly, and efficiently the study participants can solve work-related programming tasks in Java.

The participants in this study connected to my computer using Teamviewer where both IntelliJ and Visual Studio Code, Java, and GitHub Copilot (v1.86.82) were installed. Each participant was randomly assigned one task to solve with Copilot and one without it. After completing the first task, participants moved on to the second task with or without Copilot, depending on their initial assignment.

Both tasks were set up as a Java environment with JUnit integrated, allowing participants to write unit tests if desired. However, no pre-defined tests were provided, as in a real workplace environment. The study measured the time taken to solve each task to answer RQ1. To answer RQ2 I looked at the participants’ solutions to the tasks, whether they passed my tests which were not provided to the participants, and how much time each participant’s code required to run.

The tasks were estimated to take 20-30 minutes, with no time limit imposed. Participants were allowed to use all available resources such as StackOverflow
and Google with an exception for other AIs such as ChatGPT. If participants felt like they were making no progress after about 30 minutes, they could move on to the next task.

Before the study began each participant was asked to answer a survey about their general programming experience, their experience in Java, and whether they have used Copilot before. They were also provided a PDF document outlining the study’s purpose and expectations as well as a short description of how GitHub Copilot works.

To mitigate the potential effects of fatigue on the results, the participants were allocated to perform the tasks in the following manner: The first participant began with Task 1 using Copilot, while the second participant began with Task 1 without Copilot. Similarly, the third participant started with Task 2 using Copilot, and the fourth participant started with Task 2 without Copilot, and so on.

At the beginning of each session, I gave each participant a short demo of GitHub Copilot, showing how to generate code and how to view its top 10 suggestions. After solving or attempting to solve both tasks, participants answered a survey about the tasks and GitHub Copilot’s performance.

The survey consisted of seven questions, where the first three questions asked for the participant’s name, and whether he believes that his solution to Task 1 and Task 2 was correct and efficient. The remaining four questions can be seen in Figure 1.

Figure 1: The post-study survey

1. Did you find GitHub Copilot to be a helpful tool and in what way?
2. Do you think GitHub Copilot helped you to solve the tasks quicker? Please explain.
3. Do you think GitHub Copilot can be useful as a tool in a workplace environment? Do you see any future roadblocks?
4. Would you like to incorporate GitHub Copilot into your daily work? Please explain how or why not.
4.1 The Tasks

4.1.1 Task 1

Listing 1: The Validator class

```java
public class Validator {
    private final String personalNumber;

    public Validator (String personalNumber) {
        //Don’t change this!
        this.personalNumber = personalNumber;
    }

    //Create your solution below this line
    //-----------------------------------------------
    public boolean Validate () {
        //This function must validate the personalNumber of this class
        return false;
    }
}
```

The first task was to develop a validator to verify Swedish personal numbers. Participants were given a Java project that included a class named `Validator` containing an attribute called `personalNumber`, along with a constructor and a public method called `Validate`, which can be seen in [Listing 1](#). The purpose of this method was to validate the personal number of the class and return a boolean value.

The specifications and requirements of the task were provided as comments within the class, as shown on the next page.

The tests created for Task 1 use reserved personal numbers provided by the Swedish tax office[6] The purpose of the tests is to measure the correctness of the code created by the study participants. 11 different test cases were created, testing the code by, for instance, giving `null` as an input, an empty string, both incorrect and correct formats, too few digits, and too many digits among other similar examples. I also tested so that the validator returns true for every correct personal number provided by the Swedish tax office, and that it returns false for every incorrect personal number. The incorrect personal numbers were created by altering the last digit of the correct personal numbers. Next, the dates for the personal numbers were tested, checking if the participants’ solutions only accepted personal numbers of dates that exist. For instance, the 29th of February only exists in a leap year, which is one of many cases that need to be taken into account when creating a validator for personal numbers.

[6] https://www7.skatteverket.se/portal/apier-och-oppna-data/utvecklarportalen/oppetdata/Test%2C2%AD%C2%ADpersonnummer
Task 1 - Validator for a Swedish personal number (Personnummer)

A Swedish personal number has the form of yyyymmdd-abcd where yyyy defines the year, mm the month, dd the day, and abcd the last four digits where d is the verification number. For example, 19940928-8173 is an acceptable personal number.

Requirements: The class Validator must take a String (personalNumber) as its parameter and the method Validate must return a bool, this is to make it easier for me to test all the code later on.

The validator must ensure that a personal number has the correct format (yyymmdd-abcd), verify the verification digit (d) and make sure that dates that does not exist are not allowed. If the personal number is correct, validator.Validate should return true, else false.

The verification digit (d) is verified by the following algorithm:
1. Discard the first two digits.
2. Multiply every element at an even index by 2.
3. Multiply every element at an odd index by 1.
4. If the element multiplied by 2 is larger than 10, for example, 18, perform 1+8=9.
5. Do this for all the digits in the list except for the last one and add them into a sum.
6. Perform sum % 10 = r.
7. Perform 10 - r = u.
8. Perform u % 10 = t.
9. Compare t to the last digit (d) of the personal number, if it's a match that means that the verification digit is correct.

Example: 19940928-8173
1. [9, 4, 0, 9, 2, 8, 8, 1, 7, 3]
2. [18, 4, 0, 9, 4, 8, 16, 1, 14]
3. [18, 4, 0, 9, 4, 8, 16, 1, 14]
4. [1+8, 4, 0, 9, 4, 8, 1+6, 1, 1+4]
5. 9 + 4 + 0 + 9 + 4 + 8 + 7 + 1 + 5 = 47
6. 47 % 10 = 7
7. 10 - 7 = 3
8. 3 % 10 = 3
9. 3 == 3 : true

Some examples of not accepted personal numbers:

19940928-8172
19940954-8173
19941621-8173
19940014-8173
19940238-8173
4.1.2 Task 2

For the second task, the participants were given a Java project that included a skeleton code featuring a few classes and methods. This was done with the purpose of making it distinct from the previous task, with the aim of evaluating GitHub Copilot’s ability to adapt to already established code and methods.

Similar to the previous task, the purpose of this task was conveyed to the participants through comments in the code base, which can be seen below. They were required to load data into two classes, Titles and Credits, from a data set regarding HBO Max’s Movies and TV Shows[^1] and to then create two queries for this data set.

Shortened versions of the code snippets, the seven TODOs, and the hints provided to the participants can be seen in Listing 2, Listing 3, Listing 4, and Listing 5. For more details, see Listing 6, Listing 7, Listing 8, and Listing 9 in the appendix.

**Task 2**
The purpose of this task is to load data from two csv files and store it in two classes.

The first class is called Title and the second class is called Credit. The Title class contains information about a title, such as its id, title, type, and year.

The Credit class contains information about a credit, such as its person_id, id, name, character, and role. The id field in the Credit class is the same as the id field in the Title class.

**About the Data Set**
**HBO Max - TV Shows and Movies**
This data set was created to list all shows and movies available on HBO Max. It was collected from JustWatch in March 2023, containing data available in the United States.

This data set contains two files, one for the titles (titles.csv) and the other for the cast (credits.csv) of each movie and show on the platform.

This data set is a one-to-many relationship, meaning that one title can have many credits, and one credit can only have one title.

There are seven TODOs in this assignment which can be found in the classes Title, Credit, RepositoryTitles, and Main.

The file structure for Task 2 with the provided classes looked as in Figure 2

Listing 2: The most essential parts of the Credit class. For more details look in the appendix

```java
public class Credit {
    private String person_id = null;
    private String id = null;
    private String name = null;
    private String character = null;
    private String role = null;
    private Title title = null;

    public Credit(String input, RepositoryTitles repositoryTitles) {
        //TODO: Split the string by comma into an array of strings and
        //parse it into the correct fields with
        //the private method loadDataIntoCredit
        String[] data = null;
    }
}
```
Listing 3: The most essential parts of the Title class. For more details look in the appendix.

```java
public class Title {
    private String id = null;
    private String title = null;
    //...similar for the rest of the attributes...//
    private ArrayList<Credit> credits = new ArrayList<>();

    public Title(String input) {
        //TODO: Split the string by comma into an array of strings and
        // parse it into the correct fields with
        // the private method loadDataIntoTitle while ignoring rows with
        // less than 15 fields

        //Hint: There might occur some descriptions with commas in them,
        // so you need to take that into account

        String[] data = null;

        //Hint: There should be 3010 different titles in
        RepositoryTitles if done correctly
    }
}
```

Listing 4: The most essential parts of the Main class. For more details look in the appendix.

```java
public static void loadTitles(String pathToTitles, RepositoryTitles repositoryTitles) {
    //TODO: Load each row of the titles.csv file into a Title object
    // and add it to the repositoryTitles
}

public static void loadCredits(String pathToCredits, 
    RepositoryTitles repositoryTitles) {
    //TODO: Load each row of the credits.csv file into a Credit
    // object and add it to the corresponding title in the
    // repositoryTitles
}
```
Listing 5: The most essential parts of the RepositoryTitles class. For more
details look in the appendix.

```java
public void addCreditToTitle(Credit credit) {
    //TODO: Add a credit to its corresponding title
    //Hint: A credit might not have a corresponding title in the
    //repository, if so you are to discard this credit.
}

public List<Map.Entry<Title, Integer>>
    getTitlesWithMostCredits(Integer count) {
    //TODO: Create a method that returns a list of the titles with
    //the most credits together with the amount
    //of credits they have. The list should be sorted by the
    //amount of credits in descending order.
    //The key should be the title’s name (Title.title) and the
    //value should be the amount of credits.
    //The amount of <Title, Integer> pairs should be equal to
    //the count parameter.
    return null;
}

public List<Map.Entry<String, Integer>>
    topTenMostPopularActors() {
    //TODO: Create a method that returns a list with the top ten
    //most credited actors together with the amount
    //of credits they have. The list should be sorted by the
    //amount of credits in ascending order.
    //The key should be the name of the actor and the value
    //should be the amount of credits.
    return null;
}
```
For this task, seven tests were devised to assess the overall correctness of the solutions provided by the participants. The initial test is responsible for loading the data from the data set into the two classes and verifying that there are no errors. The subsequent test confirms that the data has been loaded into the two classes, and validates that the `RepositoryTitles` is not empty. Two of the tests validate that the correct amount of titles and credits are loaded into their respective classes, and another test ensures that all the credits are properly assigned to their corresponding title. Finally, the remaining two tests evaluate the accuracy of the two queries.

4.2 Pilot Study

Prior to conducting the main study, a pilot study was conducted to evaluate the feasibility of the two tasks designed as well as the general study design.

One participant was recruited for the pilot study, a computer science student with three years of programming experience in C# as well as some experience in Java. The participant was able to solve the first task which involved validating Swedish personal numbers, without Copilot, in 15 minutes, and passed 6 out of 11 tests. The second task involving the data set about HBO took 22 minutes to complete, using Copilot, with the participant passing 5 out of 7 tests.

After completing the tasks, the participant was asked to provide feedback on various aspects of the study, such as the tasks and the task instructions, the overall study design, the performance of Teamviewer and whether they felt obstructed by its latency, and the comprehensibility of the questions in the survey.

The feedback obtained from the pilot study was used to refine the instructions and procedures for the main study. For Task 2 a couple of additional `get` methods were added and the objectives of the two queries that were to be designed were changed from a HashMap to a list. Also, for Task 2 a couple of hints were added. One of the hints says that inside the description of each Title, there may occur commas that have to be taken into account when splitting each row by commas. The second hint suggests that when you are to add each `Credit` to its corresponding `Title` there may occur credits which have no corresponding `Title`, in which case said `Credit` is to be discarded. These two hints were added because the pilot study participant did not take that into account.

The initial plan for the study was to also record the screen while the participants solved the tasks to be able to go back later and see, for instance, how often the participants were to go to the browser. However, that plan was scrapped after the pilot study because the participant used the browser on his own computer which made the recording obsolete.
5 Results

<table>
<thead>
<tr>
<th>Participant</th>
<th>Time (Task 1)</th>
<th>Tests passed (Task 1)</th>
<th>Time (Task 2)</th>
<th>Tests passed (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>45</td>
<td>2</td>
<td>70</td>
<td>DNC</td>
</tr>
<tr>
<td>d</td>
<td>17</td>
<td>4</td>
<td>125</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>21</td>
<td>8</td>
<td>74</td>
<td>DNC</td>
</tr>
<tr>
<td>g</td>
<td>20</td>
<td>6</td>
<td>90</td>
<td>4</td>
</tr>
<tr>
<td>Average</td>
<td>26</td>
<td>45%</td>
<td>90</td>
<td>21%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Participant</th>
<th>Time (Task 1)</th>
<th>Tests passed (Task 1)</th>
<th>Time (Task 2)</th>
<th>Tests passed (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>43</td>
<td>3</td>
<td>77</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>32</td>
<td>5</td>
<td>73</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>73</td>
<td>9</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>37</td>
<td>4</td>
<td>79</td>
<td>3</td>
</tr>
<tr>
<td>Average</td>
<td>46</td>
<td>48%</td>
<td>68</td>
<td>46%</td>
</tr>
</tbody>
</table>

Table 2: Individual completion time in minutes of each participant together with the number of tests passed with the average values in the bottom row. Task 1 has 11 test cases and Task 2 has 7 test cases. Code that did not compile is marked with DNC.

5.1 Quantitative

To evaluate the results of this study, and to answer RQ1, a quantitative method was chosen. Participants using Copilot completed Task 1 at a much faster pace, only taking an average of 26 minutes to complete the task whereas those without Copilot needed an average of 46 minutes to complete it. However, their code was of slightly lower quality only passing 45% of the tests while those without Copilot passed 48%. Table 2 shows individual and average completion times as well as the amount of tests passed and Table 1 shows each participant’s previous programming experience.

As for Task 2, two out of four participants’ code not using Copilot did not compile and therefore passed 0 tests. Participant d required 125 minutes to complete the task while only passing two out of the seven tests, and participant g passed four of the tests, taking 90 minutes to complete the task. This results in an average of 90 minutes to complete the task and only an average of 21% of the tests passed whereas those using Copilot have an average completion time of 68 minutes and 46% of the tests passed, as can be seen in Table 2. The overall mean difference between the completion time for Task 2 is 22 minutes and the
mean difference between the tests passed is 25%.

Combining the average result of those solving Task 1 with Copilot and those solving Task 2 with Copilot you get an average completion time of 94 minutes while passing 46% of the tests. Whereas, for those without Copilot the average completion time was 136 minutes, passing 38% of the tests. Participants using Copilot completed the tasks 42 minutes faster than those not utilizing Copilot. Additionally, Copilot users achieved a higher success rate of 8% in the tests, resulting in an approximate 20% increase in overall success rate.

5.2 Qualitative

Due to such few participants actually completing the tasks correctly a qualitative method was chosen to answer RQ2 regarding the written code’s efficiency as only four of the eight participants managed to successfully validate correct personal numbers for Task 1 and nobody managed to correctly complete Task 2. Out of the four participants correctly validating personal numbers, two of them used Copilot and two of them did not use Copilot, and as for the code’s efficiency, there was no significant difference between them.

Task 1  An example of an efficient solution to Task 1 will be listed below, which will be used for evaluating the efficiency of the participants’ solution.

1. Confirm that the input is not null.
2. Confirm that the input has the right format by using a regular expression, and then parse it into the DateTimeFormat
3. Discard the first two digits, the hyphen, and the last digit of the personal number while also saving the last digit for later.
4. Multiply every digit at an even index by two, and if the digit that was multiplied by two becomes larger than 9, perform the algorithmic operation by using modulo.
5. Take the sum of every element in the list, perform the arithmetic operations, and compare it to the last digit that was saved in step 3.

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html
Participant | Task 1 - Copilot
--- | ---
a | No step 1, did all of the other steps accordingly except for step 2 where he used the older, less efficient `SimpleDateFormat`.
d | No steps 1,2. Did step 3 accordingly, used the less efficient `string.substring` operations for steps 4,5.
f | Did all five steps accordingly.
g | No step 1, used a regular expression for step 2 but did not parse it into the `DateTimeFormat`. Did steps 3,4,5 accordingly.

| Participant | Task 1 - Without Copilot
--- | ---
b | Only did steps 4,5 accordingly.
c | No step 1, attempted steps 2,3 but in the wrong order. Did steps 4,5 accordingly.
e | No step 1, attempted step 2 but did not use the `DateTimeFormat`. Did step 3 less efficiently and did steps 4,5 accordingly.
h | No steps 1,2, did steps 3,5 accordingly. Did step 4 in a much less efficient manner.

Table 3: A short comparison of the efficiency of each participant’s solution to Task 1, compared to the solution listed above.

Among all eight participants who attempted to solve Task 1, the four participants using Copilot managed to complete a combined total of 12 steps, compared to the four participants not using Copilot who reached a total of eight steps passed, as can be seen in Table 3. Only one of the eight participants, participant f, checked if the input was `null`. Participant f was also the only one out of the eight participants who accounted for all five steps in his solution, and he used Copilot. Also to be noted is that all four participants who used Copilot did step 3 accordingly, compared to those not using Copilot where participant h was the only one who did it. As for the participants solving Task 1 without Copilot, all four of them managed to perform the arithmetic operations in step 5 in an efficient manner. While three participants successfully completed step 4 with efficiency, participant h approached it differently. Instead of checking if the index of the digit was even, participant h directly compared it to multiple even numbers. Those were however the only parts of their solutions that were done efficiently although participant c performed steps 2 and 3 accordingly but in the wrong order, making it both incorrect and less efficient.

Task 2 An example of an efficient solution to Task 2, excluding the two queries, will be listed below, which will be used for evaluating the efficiency of the participants’ solution.
Table 4: A short comparison of the efficiency of each participant’s solution to Task 2, compared to the solution listed above.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Task 2 - Copilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Did steps 1,2,3,4 accordingly. No step 5.</td>
</tr>
<tr>
<td>c</td>
<td>Did steps 1,2,3, attempted step 4 but did not account for arrays with less than 15 elements. No step 5.</td>
</tr>
<tr>
<td>e</td>
<td>Did steps 1,3,5 accordingly, steps 2,4 less efficiently.</td>
</tr>
<tr>
<td>h</td>
<td>Did steps 1,2,3,5 accordingly. Did step 4 but did not account for commas within quotation marks.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Participant</th>
<th>Task 2 - Without Copilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>No steps 1,2. Did steps 3,5 correctly, also did step 4 but did not account for commas within quotation marks.</td>
</tr>
<tr>
<td>d</td>
<td>Did step 1 less efficiently by using a <code>StringBuilder</code>(^{12}). Did steps 2,3,5 accordingly, also did step 4 but did not account for commas within quotation marks.</td>
</tr>
<tr>
<td>f</td>
<td>Did steps 1,2,3,5 accordingly, did not account for commas within quotation marks for step 4.</td>
</tr>
<tr>
<td>g</td>
<td>Did step 1 by using the less efficient <code>Stream</code>(^{14}) class. Did steps 2,3,5 accordingly, also did step 4 but did not account for commas within quotation marks.</td>
</tr>
</tbody>
</table>

1. Use the `BufferedReader`\(^{11}\) class to read each row from the files.

2. Parse each row into the `Title` and `Credit` class.

3. In the `Credit` class, split each row by comma into an array of strings by using the `String.split`\(^{12}\) method.

4. In the `Title` class, split each row by comma into an array of strings while also ignoring the commas appearing within quotation marks by using the `String.split` method. After doing so, confirm that the array has 15 elements.

5. In the `RepositoryTitles` class, add each `Credit` to its corresponding `Title` by first confirming that said `Title` exists.

The four participants using Copilot for Task 2 passed a combined total of 14 steps, compared to the four not using Copilot who passed 12 steps, as can be seen in Table 4. As for the four participants using Copilot for Task 2, all four of them used the `BufferedReader` class to read each row from the files in an efficient manner, compared to the four not using Copilot where `participant f` was the

\(^{11}\) [https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html)

\(^{12}\) [https://www.baeldung.com/string/split](https://www.baeldung.com/string/split)
only one who did it. All eight participants managed to do step 3 accordingly but only one, participant b, accounted for step 4. And as for step 5, participants e and h were the only ones out of the four participants using Copilot that confirmed that a Credit had a corresponding Title before adding it to its Title. In contrast, all four participants who did not use Copilot accounted for this. Also, to be taken into consideration is that participant a ran out of time and did therefore not attempt steps 1 and 2.

The queries inside the RepositoryTitles class were ignored for this part of the results as nobody managed to solve them correctly and therefore any efficiency in the code will be hard to judge.

5.3 Survey Evaluation

In the post-study survey, which can be seen in Figure 1, six out of eight participants found Copilot to be a helpful tool: "Copilot was very useful. It managed to give me a decent solution based on step-by-step pseudo code. I could also avoid any syntax issues coming from being used to another programming language. This way I could read between the lines what the code actually did even if I’m used to a different syntax" (participant d). Participant e thought that it was very useful for creating a code base: "Yes, I think it was useful when creating the code base. Also, I could add more advanced functionality using the commenting function".

Two participants did not find Copilot to be as useful, partly due to having higher expectations of it and also thinking that it mostly just generated code that he was already about to type: "Mostly it entered what I was about to type. As an auto-complete feature it did well, but I probably would not use it to generate entire methods as fiddling around in the GUI would break the flow" (participant a).

For the next question, seven out of the eight participants stated that they believe that Copilot helped them to solve the tasks faster: "More often than not, Copilot could anticipate what the next line of code would be, speeding up the basic tasks. And also as long as I had an idea of what I wanted to do, solving the problem became very fluent, and once stuck it would give several suggestions" (participant d). Whereas participant a stated that the impact of Copilot was probably enhanced due to Teamviewer’s latency affecting his workflow. Participant b did not find Copilot as helpful: "I had difficulties to understand how to use Copilot in IntelliJ. With more knowledge about the tool I think it could improve coding time".

For the last major question of the post-study survey: "Do you think GitHub Copilot can be useful as a tool in a workplace environment? Do you see any future roadblocks?" all eight participants answered yes and three of them focused on privacy issues as a future roadblock: "At most companies, I think it will be
an issue that your own code is used by Copilot for learning and can be shared outside of the company. This is a major drawback!” (participant e). Participant a states that it could definitely be useful for novice programmers: “I can certainly see its uses, suggested solutions can help novice developers learn new ways to solve problems, generally help in learning new languages and shorten the time spent typing simpler code snippets”. Another concern that was raised is that Copilot could potentially hinder programmers’ learning curve: “Also I’m not sure how it would affect developers’ own development in the long run, maybe a slower learning curve” (participant d).

6 Discussion

The result of this study indicates that the tasks presented to the participants turned out to be much more difficult than anticipated, making it difficult to measure performance and efficiency. Task 1 had a more reasonable time completion, with an average of 26 minutes with Copilot and 46 minutes without Copilot where participant e is an outlier, affecting the average completion time by a lot, as can be seen in Table 2. What we can observe for Task 1 is that participants using Copilot passed 3% fewer tests compared to those not using Copilot. This could be because the participants (b, c, e, and h) solving Task 1 without Copilot had significantly more experience than those using Copilot (a, d, f, and g), as can be seen in Table 1. The amount of tests passed by the participants turned out to be much lower than expected, most likely due to the tasks taking up a lot of time, but also due to choices I made when designing the tasks. The idea was that the tasks were supposed to mimic a workplace environment, where a customer wants a validator for a Swedish personal number created. Therefore, no test cases were provided to the participants as that would typically not be provided in the workplace scenario. This was for some reason not explicitly mentioned in the instructions for the participants, which in combination with not providing test cases could explain why most participants did not take into account, for instance, null as an input, or an empty string, resulting in a low amount of tests passed.

As for Task 2, the time anticipated to complete it turned out to be much too low, and both filling out the empty methods of the classes as well as creating the two queries turned out to be too much for this study. Nobody was able to complete both of the queries and nobody was able to correctly parse the data set into the appropriate classes. I believe this was partly because most of the participants were more experienced with the programming language C# rather than Java. But also, because the task was large, with a lot of text to describe the data set, the class structure, and the purpose of the task, which takes time to read and comprehend. The data set also turned out to be difficult to handle, with some rows not having the proper amount of fields and some descriptions including commas inside them, making it difficult to properly separate each field by a comma. From my observations, this was the major obstacle to this task. I
also believe that not providing test cases affected the participants’ performance for this task. Despite this, the participants using Copilot completed the task much faster, with an average completion time of 68 minutes, compared to those not using Copilot who had an average completion time of 90 minutes. They also had a much higher success rate in passing the tests, 46% compared to the 21% success rate of those not using Copilot, as can be seen in Table 2.

The qualitative analysis of the written code’s efficiency showed that the use of Copilot did not necessarily lead to more or less efficient code. The efficiency of the participants’ code was difficult to fairly judge due to such a low amount of tests passed, and for the tests that passed no significant difference in run-time can be seen. The combination of the participants using Copilot for Task 1 and Copilot for Task 2 passed a combined total of 26 steps, compared to the ones not using Copilot which passed 20 steps. The group using Copilot clearly passed more of the steps for the efficiency analysis of their code, with an average of 6.5 steps passed per participant, compared to those not using Copilot which had an average of five steps passed per participant. However, due to not being able to quantitatively assess the code, the difference is not significant enough to draw any conclusions.

In the post-study evaluation, most participants were generally positive about Copilot whereas participant b who was more skeptical had some issues with how to use Copilot. Some of the drawbacks mentioned by the participants regarding privacy issues of Copilot, by for instance participant e in the survey evaluation section is based on an incorrect assumption that Copilot learns from the user’s code. And while Copilot can learn from the user’s code, this function can easily be disabled in Copilot’s user settings, therefore eliminating this potential issue [1]. Question four was not used for the results as the question was very similar to question three and did therefore not generate any interesting answers.

Observations I made during the participants solving of the tasks is that Copilot has a steeper learning curve than what I expected. I noticed that some participants rarely used Copilot to generate code, if at all, and also as was conveyed from the post-study survey, at least one participant had problems with understanding how to generate code with Copilot. From what I observed only participant f, who was the only participant with previous experience with Copilot, actually used Copilot to its full potential, using the Copilot toolbar which shows the top 10 suggested solutions generated by Copilot. Participant f was also the only one who accounted for all five steps in his solution to Task 1, for which he used Copilot. What I also observed is that generally, it seemed like participants with less programming experience used Copilot to greater effect than those with more experience.

Another observation is that most participants never compiled their code. This came as a surprise for me as I made sure to ask the participants beforehand in the pre-study survey if they preferred Visual Studio Code or IntelliJ for Java
and made sure to have everything properly installed. This might be because I did not provide any test cases and the participants not compiling their code could also explain why the number of tests passed was so low.

Another aspect that could have affected the participants’ performance is that they could be affected by my observing, potentially judging them, as they code. I also observed that some participants did not completely understand the instructions for Task 2, misunderstanding the structure of the classes and each class’ purpose, which resulted in them writing unnecessary code in the wrong place, costing a lot of time and potentially causing some frustration. For instance, participant e first missed that the two entity classes provided, Title and Credit, existed. In the end, he realized that there was another class he had missed, RepositoryTitles, which upon discovery he decided to give up.

Some of the previously discussed shortcomings of this study can be explained by the design of the study. Generally, the participants might have been more comfortable solving the tasks on their own computers on their own time rather than on my computer through Teamviewer. However, this could introduce some other difficulties, each participant would have to install Java on their own computer and they would also have to create their own GitHub Copilot subscription, which has a free trial but still requires you to provide payment information. Perhaps any future studies could set up a virtual machine with everything installed and ready to go, letting the participants solve the tasks when they have time, although this would introduce the possibility of the participants providing the wrong amount of time required for them to complete the tasks. Another study design for a future study would be to have everything set up on a physical computer rather than virtually, allowing the participants to solve the tasks in an isolated room without fear of being observed.

Ideally, these problems would have been identified and fixed during the pilot study, but due to not having many potential users for the user study only one person was chosen for the pilot study which in this case resulted in a very different result for the pilot study compared to the actual study. One potential reason for the difference in performance between the pilot study and the study is that the pilot study participant recently graduated and therefore might be more accustomed to these types of tasks.

Things that I would change if I were to make this study again would be to make the tasks significantly smaller and more concise by providing test cases which would hopefully make it easier to recruit participants as well as measure the results.
7 Conclusion

In conclusion, the quantitative results of this study indicate that the use of Copilot can significantly reduce the completion time of programming tasks, as participants using Copilot completed Task 1 and Task 2, on average, 20 and 22 minutes faster than those without Copilot, respectively. And while those completing Task 1 with Copilot had a somewhat lower success rate in passing the tests than those without Copilot, they also had significantly less experience in programming. By combining the participants solving Task 1 with Copilot and the ones solving Task 2 with Copilot the difference in experience between the two groups was mitigated. This shows that the group using Copilot had both a faster completion time as well as a higher success rate in passing the tests compared to the group not using Copilot.

The qualitative analysis of the written code’s efficiency showed that the use of Copilot did not necessarily lead to more or less efficient code, as there was no significant difference between the code written by participants using Copilot and those who did not use Copilot.

In general, it seems that Copilot can be a useful tool for novice programmers who may benefit from its ability to generate code and provide helpful suggestions rather than getting stuck with semantics or syntax. More experienced programmers may also benefit from Copilot, making it easier to write code in a programming language with unfamiliar syntax.

There are certain limitations to this study that should be taken into consideration. Firstly, the sample size was relatively small, which may limit the generalizability of the findings. Additionally, the participants’ success rates in generating solutions to the tasks were notably low, particularly for Task 2. Furthermore, Copilot had a somewhat steep learning curve, which could potentially decrease its usefulness for certain participants. Additionally, the study only examined the use of Copilot for Java, so it is unclear whether the results would generalize to other programming languages. Future research could explore the use of Copilot for other programming languages, perhaps in a more work-related environment, allowing the participants to get familiar with Copilot over time rather than just a couple of hours.

Overall, this study provides insight into the potential benefits and limitations of using Copilot as a code-generation tool for programming tasks. By understanding the strengths and weaknesses of Copilot, programmers can make informed decisions about whether to use this tool and how to use it effectively.
References


Listing 6: The Credit class

```java
public class Credit {
    private String person_id = null;
    private String id = null;
    private String name = null;
    private String character = null;
    private String role = null;
    private Title title = null;

    public Credit(String input, RepositoryTitles repositoryTitles) {
        //TODO: Split the string by comma into an array of strings and
        //parse it into the correct fields with
        //the private method loadDataIntoCredit
        String[] data = null;
    }

    private void loadDataIntoCredit(String[] data, RepositoryTitles repositoryTitles) {
        this.person_id = data[0];
        this.id = data[1];
        this.name = data[2];
        this.character = data[3];
        this.role = data[4];
        this.title = repositoryTitles.getTitleFromRepository(this.id);
    }
}
```

A Appendix

1. The Credit class excluding package declarations, imports, and a couple of get and print methods looked as in Listing 6.

2. The Title class excluding package declarations, imports, and a couple of get and print methods looked as in Listing 7.

3. The Main class excluding the task description, package declarations, and imports looked as in Listing 8.

4. The RepositoryTitles class excluding package declarations, imports, and a couple of get and print methods looked as in Listing 9.
public class Title {
    private String id = null;
    private String title = null;
    //...similar for the rest of the attributes...//
    private ArrayList<Credit> credits = new ArrayList<>();

    public Title(String input) {
        //TODO: Split the string by comma into an array of strings and
        // parse it into the correct fields with
        // the private method loadDataIntoTitle while ignoring rows with
        // less than 15 fields
        //Hint: There might occur some descriptions with commas in them,
        // so you need to take that into account
        String[] data = null;

        //Hint: There should be 3010 different titles in
        RepositoryTitles if done correctly
    }

    private void loadDataIntoTitle(String[] data) {
        this.id = data[0] != null ? data[0] : null;
        //...similar for the rest of the attributes...//
        this.credits = new ArrayList<>();
    }

    public void addCredit(Credit credit) {
        this.credits.add(credit);
    }
}
Listing 8: The Main class

/*
This class is the main class of the application.
It is responsible for loading the data from the csv files.

The main method is the entry point of the application.
It is responsible for creating an instance of the RepositoryTitles
class and calling the loadTitles and loadCredits methods.
*/

class Main {

    public static void main(String[] args) throws FileNotFoundException {
        String pathToTitles = "C:\Users\IsakNilsson\Documents\Examensarbete\Task2\titles.csv";
        String pathToCredits = "C:\Users\IsakNilsson\Documents\Examensarbete\Task2\credits.csv";

        RepositoryTitles repositoryTitles = new RepositoryTitles();

        loadTitles(pathToTitles, repositoryTitles);
        loadCredits(pathToCredits, repositoryTitles);
    }

    public static void loadTitles(String pathToTitles, RepositoryTitles repositoryTitles) {
        //TODO: Load each row of the titles.csv file into a Title object
        // and add it to the repositoryTitles
    }

    public static void loadCredits(String pathToCredits,
        RepositoryTitles repositoryTitles) {
        //TODO: Load each row of the credits.csv file into a Credit
        // object and add it to the corresponding title in the
        // repositoryTitles
    }
}
public class RepositoryTitles {
    private Map<String, Title> repositoryTitles;

    public RepositoryTitles() {
        this.repositoryTitles = new HashMap<>();
    }

    public void addTitle(Title title) {
        this.repositoryTitles.put(title.getId(), title);
    }

    public void addCreditToTitle(Credit credit) {
        //TODO: Add a credit to its corresponding title
        //Hint: A credit might not have a corresponding title in the
title repository, if so you are to discard this credit.
    }

    public List<Map.Entry<Title, Integer>> getTitlesWithMostCredits(Integer count) {
        //TODO: Create a method that returns a list of the titles with
        //the most credits together with the amount
        //of credits they have. The list should be sorted by the
        //amount of credits in descending order.
        //The key should be the title’s name (Title.title) and the
        //value should be the amount of credits.
        //The amount of <Title, Integer> pairs should be equal to
        //the count parameter.
        return null;
    }

    public List<Map.Entry<String, Integer>> topTenMostPopularActors() {
        //TODO: Create a method that returns a list with the top ten
        //most credited actors together with the amount
        //of credits they have. The list should be sorted by the
        //amount of credits in ascending order.
        //The key should be the name of the actor and the value
        //should be the amount of credits.
        return null;
    }
}